knn

July 16, 2025

[17]:
```
# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'CS231n/assignments/assignment1'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
/home/doubeecat/CS231n/assignments/assignment1/cs231n/datasets
/home/doubeecat/CS231n/assignments/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

[18]:
```
# Run some setup code for this notebook.

import random
```

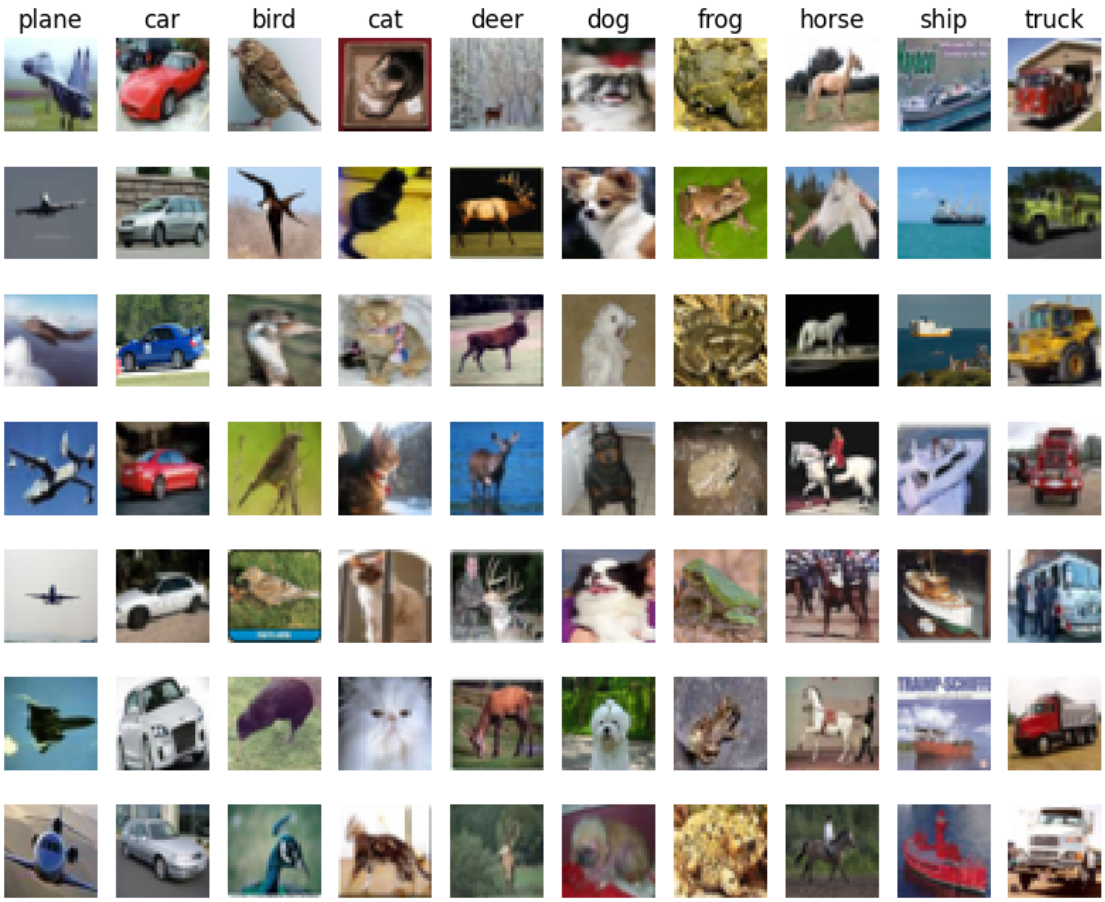
```
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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
```

[19]:
```
# 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)
```

```
Clear previously loaded data.

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

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



[21]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

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

[22]:
```
from cs231n.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

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

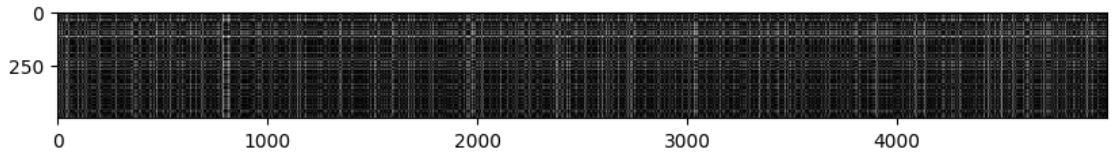
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

[28]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

[24]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

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

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

Your Answer : Because those data in bright column is more distant in the L2 distance. It may be a outlier such as a photo of car get into the dataset of cats. Or it might be some features(like lights) in photo.

For the columns, some of the samples are unique (like black cat in white cats).

[29]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

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

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

You should expect to see approximately 27% accuracy. Now lets try out a larger `k`, say `k = 5`:

[33]:
```
y_test_pred = classifier.predict_labels(dists, k=10)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

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

YourAnswer : 1,2,3

YourExplanation :

1. Because the fomula of L1 distance is $|a_{i,j} - b_{i,j}|$, so adding the μ won't change the value.
2. Same with the first step.
3. May have the effcet. Because $D' = \frac{1}{\sigma} D$, but the KNN is depends on the sort of value, so it won't have an efffect.
4. $D' = \sum \frac{|a_{i,j} - b_{i,j}|}{\sigma_{i,j}}$ so it will have an effect.
5. Rotating the axis will get the distance changed.

[35]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

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

```
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

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

[38]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

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

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

[40]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took␣
 ↪to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

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

```
Two loop version took 13.232369 seconds
One loop version took 12.852840 seconds
No loop version took 0.148510 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.

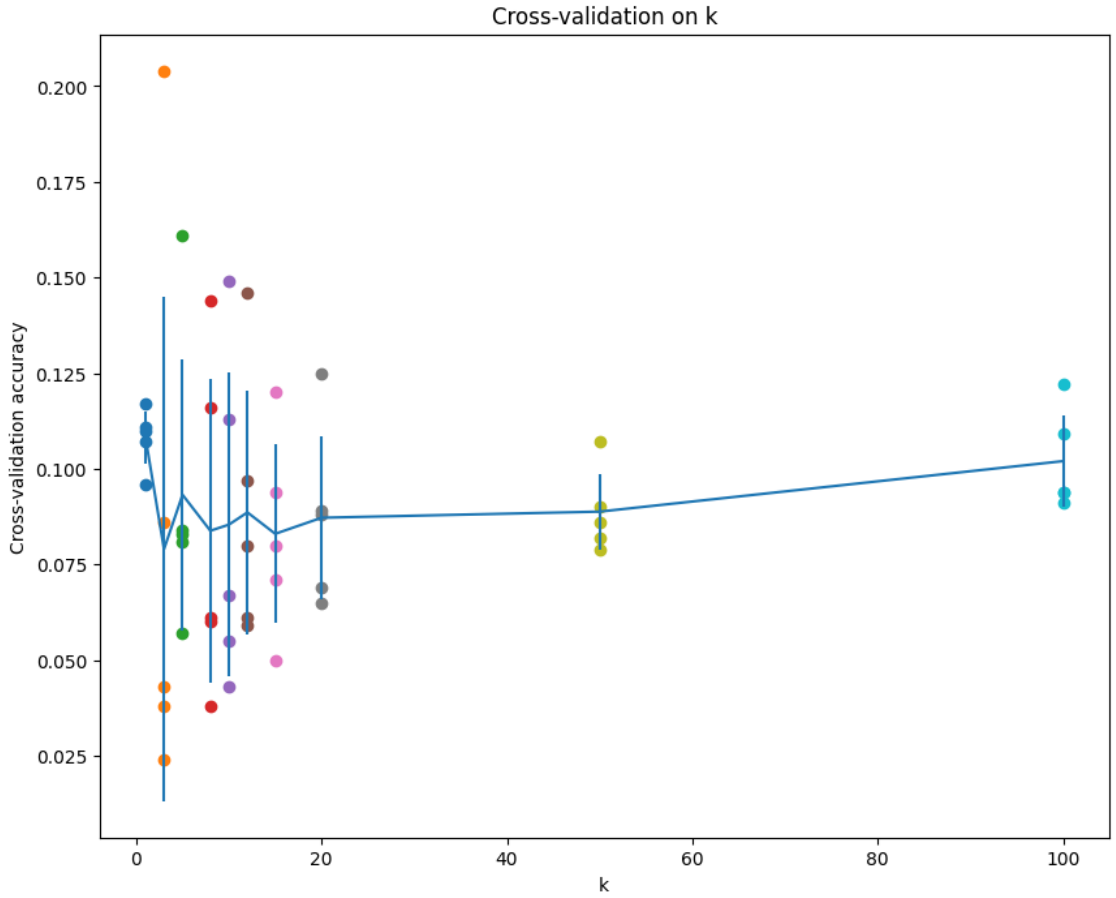
[48]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []

X_train_folds = np.array_split(X_train,num_folds)
y_train_folds = np.array_split(y_train,num_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.                                #
################################################################################


# 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 = {k:[] for k in k_choices}

for now_k in sorted(k_to_accuracies):
    for chos in range(num_folds):
        X_new_train = np.concatenate(X_train_folds[:chos] +␣
 ↪X_train_folds[chos+1:])
        y_new_train = np.concatenate(y_train_folds[:chos] +␣
 ↪y_train_folds[chos+1:])
        classifier.train(X_new_train,y_new_train)

        nowk_results = classifier.predict_labels(X_train_folds[chos],now_k)
        accuracy = np.mean(y_new_train[chos] == nowk_results)
        k_to_accuracies[now_k].append(accuracy)
################################################################################
```

```
# 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.                        #
#########################################################################


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

```
k = 1, accuracy = 0.117000
k = 1, accuracy = 0.110000
k = 1, accuracy = 0.107000
k = 1, accuracy = 0.111000
k = 1, accuracy = 0.096000
k = 3, accuracy = 0.038000
k = 3, accuracy = 0.043000
k = 3, accuracy = 0.024000
k = 3, accuracy = 0.086000
k = 3, accuracy = 0.204000
k = 5, accuracy = 0.084000
k = 5, accuracy = 0.083000
k = 5, accuracy = 0.057000
k = 5, accuracy = 0.081000
k = 5, accuracy = 0.161000
k = 8, accuracy = 0.061000
k = 8, accuracy = 0.060000
k = 8, accuracy = 0.038000
k = 8, accuracy = 0.116000
k = 8, accuracy = 0.144000
k = 10, accuracy = 0.067000
k = 10, accuracy = 0.055000
k = 10, accuracy = 0.043000
k = 10, accuracy = 0.113000
k = 10, accuracy = 0.149000
k = 12, accuracy = 0.080000
k = 12, accuracy = 0.061000
k = 12, accuracy = 0.059000
k = 12, accuracy = 0.097000
k = 12, accuracy = 0.146000
k = 15, accuracy = 0.080000
k = 15, accuracy = 0.071000
k = 15, accuracy = 0.050000
k = 15, accuracy = 0.094000
```

```
k = 15, accuracy = 0.120000
k = 20, accuracy = 0.069000
k = 20, accuracy = 0.089000
k = 20, accuracy = 0.065000
k = 20, accuracy = 0.088000
k = 20, accuracy = 0.125000
k = 50, accuracy = 0.079000
k = 50, accuracy = 0.086000
k = 50, accuracy = 0.082000
k = 50, accuracy = 0.090000
k = 50, accuracy = 0.107000
k = 100, accuracy = 0.094000
k = 100, accuracy = 0.109000
k = 100, accuracy = 0.094000
k = 100, accuracy = 0.122000
k = 100, accuracy = 0.091000
```

[49]:
```
# 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()
```



[53]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 6

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

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

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

Inline Question 3

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

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

YourAnswer : 2,4

YourExplanation :

1. Wrong. For example, the boundary of k-NN will be complex when k = 1.
2. Right. k = 1 will always fit the training dataset.k = 5 might not work well in sometimes.
3. Wrong. When k = 1, it might get over-fitting so that it can get more test error.
4. Right. Because the time complexity of classify is O(N), where N is the scale of training set.
5. Wrong.

softmax

July 16, 2025

[3]:
```
# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'CS231n/assignments/assignment1'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
/home/doubeecat/CS231n/assignments/assignment1/cs231n/datasets
/home/doubeecat/CS231n/assignments/assignment1
```

1 Softmax Classifier exercise

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

In this exercise you will:

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

[4]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
```

```
import matplotlib.pyplot as plt

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

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

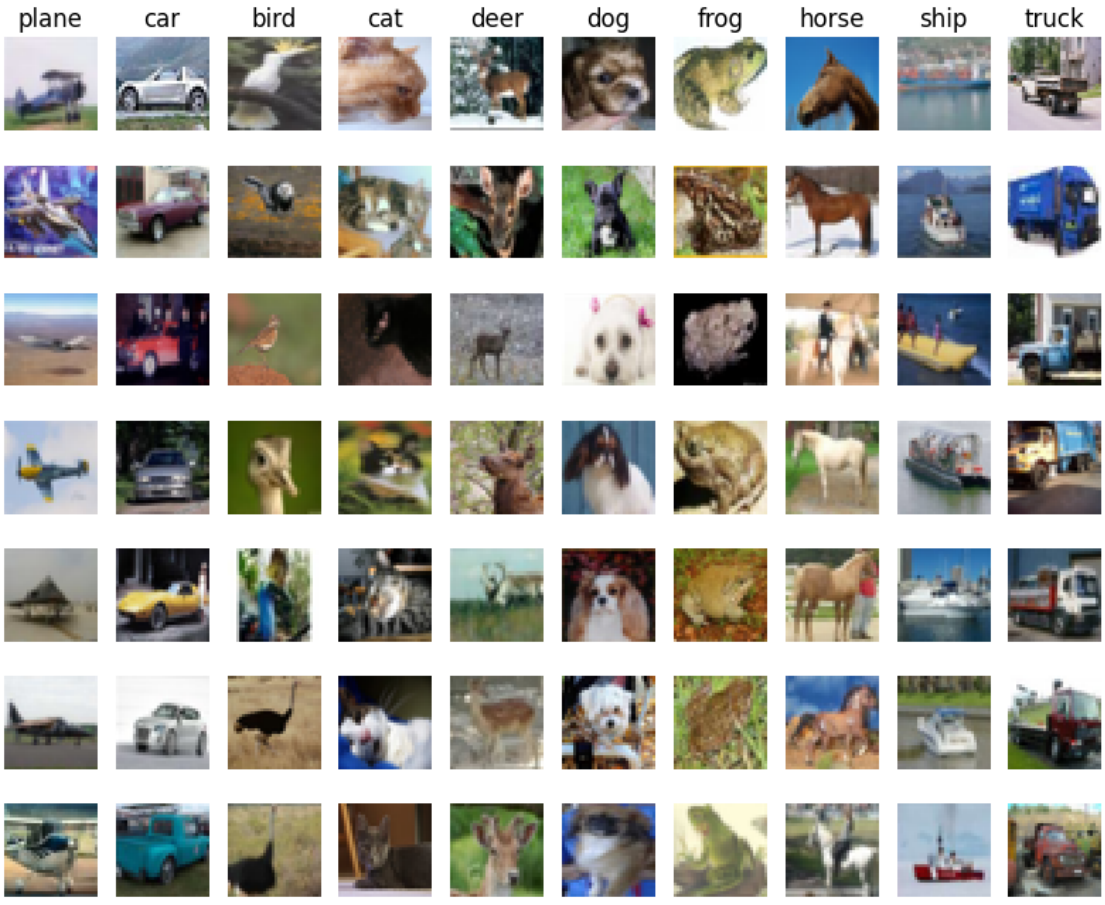
1.1 CIFAR-10 Data Loading and Preprocessing

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

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



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

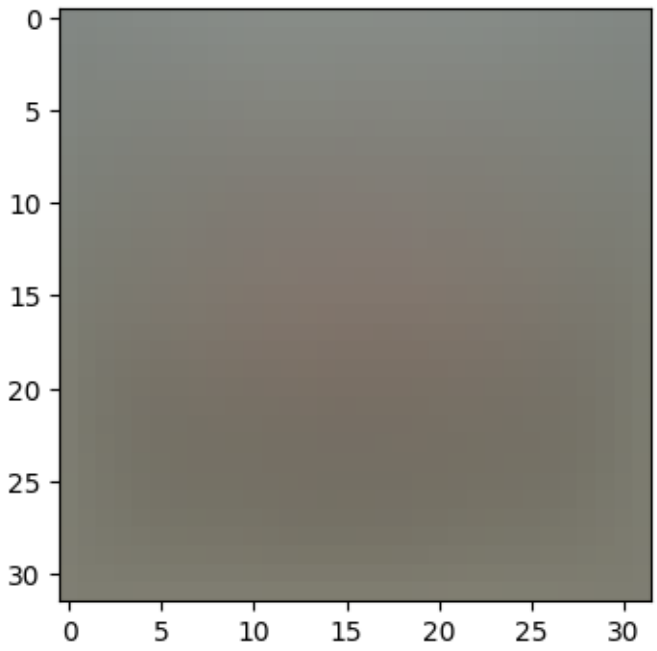
[8]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))
```

```
# 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)
```

[9]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
 ↪image
plt.show()

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

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

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

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




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

1.2 Softmax Classifier

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

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

[10]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

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

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

```
loss: 2.302818
loss: 2.302818
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer : When the model is untrained, every score is expected to be same. And every column's P is expected to be 0.1.

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

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

[11]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

```
numerical: 0.004170 analytic: 0.004170, relative error: 1.659347e-08
numerical: -0.016985 analytic: -0.016985, relative error: 9.834223e-10
numerical: -0.004205 analytic: -0.004205, relative error: 7.502158e-09
numerical: -0.000892 analytic: -0.000892, relative error: 2.874488e-08
numerical: 0.004654 analytic: 0.004654, relative error: 1.792354e-09
numerical: -0.005906 analytic: -0.005906, relative error: 8.609158e-09
numerical: -0.006721 analytic: -0.006721, relative error: 2.995462e-09
numerical: -0.000594 analytic: -0.000594, relative error: 1.623828e-07
numerical: -0.014670 analytic: -0.014670, relative error: 5.762356e-09
numerical: 0.005599 analytic: 0.005599, relative error: 1.690123e-08
numerical: 0.018592 analytic: 0.018592, relative error: 6.219887e-10
numerical: 0.008189 analytic: 0.008189, relative error: 6.236006e-09
numerical: 0.000356 analytic: 0.000356, relative error: 2.966964e-08
numerical: 0.002896 analytic: 0.002896, relative error: 3.082644e-08
numerical: 0.006749 analytic: 0.006749, relative error: 5.983892e-09
numerical: -0.008226 analytic: -0.008226, relative error: 4.153829e-09
```

```
numerical: 0.004845 analytic: 0.004845, relative error: 4.350615e-09
numerical: -0.022285 analytic: -0.022285, relative error: 5.551491e-10
numerical: -0.000567 analytic: -0.000567, relative error: 5.212403e-08
numerical: 0.005353 analytic: 0.005353, relative error: 4.522183e-09
```

Inline Question 2

Although gradcheck is reliable softmax loss, it is possible that for SVM loss, once in a while, a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a svm loss gradient check could fail? How would change the margin affect of the frequency of this happening?

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

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

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

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

YourAnswer : Because the grad of max function is non-differentiable. So it's an math error? But I think it will only affect some points on the bound. When the delta of correct score and the wrong score is exactly δ, then the gradiant of max is unstable.

[12]:
```
# Next implement the function softmax_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
Naive loss: 2.302818e+00 computed in 5.710632s
Vectorized loss: 2.413398e+00 computed in 0.018360s
difference: -0.110580
```

[13]:
```
# Complete the implementation of softmax_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.
```

```
import time
# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

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

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

```
Naive loss and gradient: computed in 5.504014s
Vectorized loss and gradient: computed in 0.008897s
difference: 348.236208
```

1.2.1 Stochastic Gradient Descent

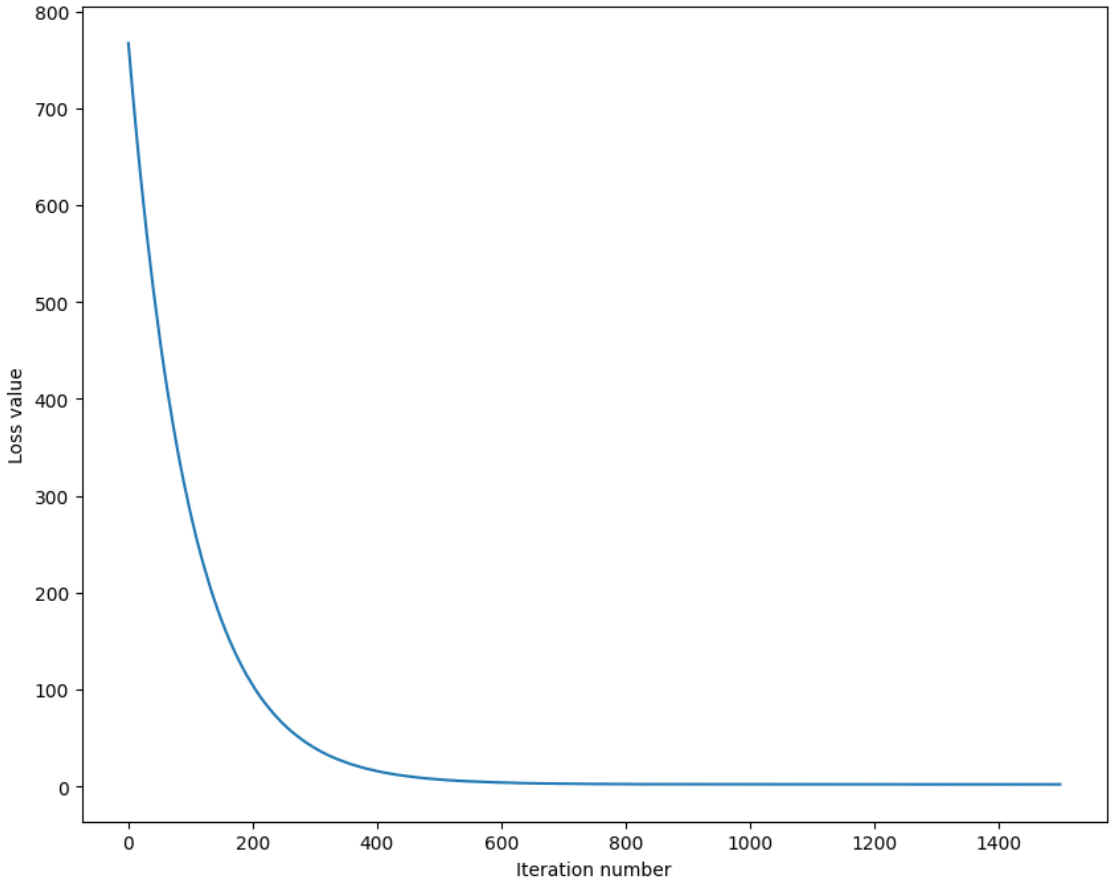
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[14]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import Softmax
softmax = Softmax()
tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 766.955577
iteration 100 / 1500: loss 281.631100
iteration 200 / 1500: loss 104.411583
iteration 300 / 1500: loss 39.487911
iteration 400 / 1500: loss 15.708160
iteration 500 / 1500: loss 7.085774
iteration 600 / 1500: loss 3.916972
iteration 700 / 1500: loss 2.740739
iteration 800 / 1500: loss 2.378363
iteration 900 / 1500: loss 2.174706
```

```
iteration 1000 / 1500: loss 2.146270
iteration 1100 / 1500: loss 2.143745
iteration 1200 / 1500: loss 2.145690
iteration 1300 / 1500: loss 2.079422
iteration 1400 / 1500: loss 2.099851
That took 4.484080s
```

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



[16]:
```
# Write the LinearClassifier.predict function and evaluate the performance on
# both the training and validation set
# You should get validation accuracy of about 0.34 (> 0.33).
y_train_pred = softmax.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
```

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

```
training accuracy: 0.330490
validation accuracy: 0.343000
```

[17]:
```
# Save the trained model for autograder.
softmax.save("softmax.npy")
```

```
softmax.npy saved.
```

[29]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.365 (> 0.36) on the validation set.

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

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

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-3, 1e-6,1e-5,1e-4]
regularization_strengths = [2.5e4, 1e4,1e3,5e4]
best_lr,best_reg = 0,0
```

```
for now_lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, now_lr, reg,num_iters=200,␣
 ↪verbose=True) #   100?
        y_train_pred = softmax.predict(X_train)
        tr_acc = np.mean(y_train == y_train_pred)
        y_val_pred = softmax.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)
        results[(now_lr,reg)] = (tr_acc,val_acc)
        if val_acc > best_val:
            best_val = val_acc
            best_lr,best_reg = now_lr,reg



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

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

```
iteration 0 / 200: loss 790.162697
```

```
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:81:
RuntimeWarning: divide by zero encountered in log
  cp = -np.log(prob[np.arange(num_train),y])
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:82:
RuntimeWarning: overflow encountered in scalar multiply
  loss = np.sum(cp) / num_train + reg * np.sum(W * W)
/home/doubeecat/CS231n/.venv/lib/python3.12/site-
packages/numpy/_core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:82:
RuntimeWarning: overflow encountered in multiply
  loss = np.sum(cp) / num_train + reg * np.sum(W * W)
```

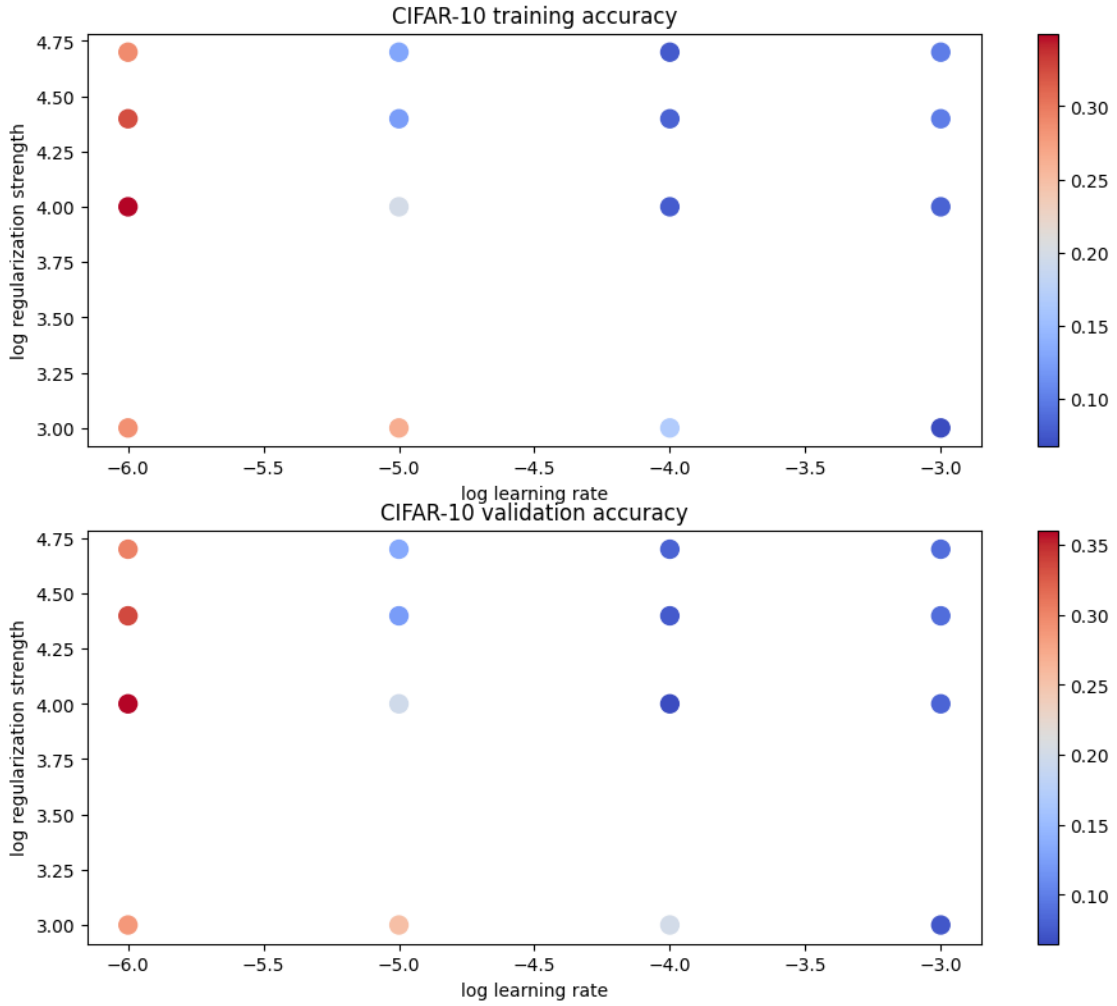
```
iteration 100 / 200: loss inf
```

```
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:88:
RuntimeWarning: overflow encountered in multiply
  dW = np.dot(X.T,ds) + 2 * reg * W
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:76:
RuntimeWarning: overflow encountered in dot
  scores = X.dot(W)
```

```
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:76:
RuntimeWarning: invalid value encountered in dot
  scores = X.dot(W)
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:77:
RuntimeWarning: overflow encountered in subtract
  scores -= np.max(scores,axis = 1,keepdims=True)
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:77:
RuntimeWarning: invalid value encountered in subtract
  scores -= np.max(scores,axis = 1,keepdims=True)

iteration 0 / 200: loss 311.808713
iteration 100 / 200: loss inf
iteration 0 / 200: loss 37.146297
iteration 100 / 200: loss inf
iteration 0 / 200: loss 1542.827560
iteration 100 / 200: loss inf
iteration 0 / 200: loss 775.134960
iteration 100 / 200: loss 2.131320
iteration 0 / 200: loss 312.431672
iteration 100 / 200: loss 7.288667
iteration 0 / 200: loss 36.479294
iteration 100 / 200: loss 22.846759
iteration 0 / 200: loss 1562.577322
iteration 100 / 200: loss 2.177819
iteration 0 / 200: loss 774.253488
iteration 100 / 200: loss 9.425512
iteration 0 / 200: loss 308.877640
iteration 100 / 200: loss 4.742315
iteration 0 / 200: loss 36.330074
iteration 100 / 200: loss 4.569262
iteration 0 / 200: loss 1546.112551
iteration 100 / 200: loss 17.052509
iteration 0 / 200: loss 777.540734
iteration 100 / 200: loss inf
iteration 0 / 200: loss 312.258562
iteration 100 / 200: loss inf
iteration 0 / 200: loss 36.482407
iteration 100 / 200: loss 55.672110
iteration 0 / 200: loss 1550.311035
iteration 100 / 200: loss inf
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.284714 val accuracy: 0.287000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.349633 val accuracy: 0.360000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.323980 val accuracy: 0.335000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.288265 val accuracy: 0.303000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.264102 val accuracy: 0.254000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.198184 val accuracy: 0.198000
lr 1.000000e-05 reg 2.500000e+04 train accuracy: 0.123061 val accuracy: 0.122000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.129245 val accuracy: 0.134000
```

```
lr 1.000000e-04 reg 1.000000e+03 train accuracy: 0.166776 val accuracy: 0.200000
lr 1.000000e-04 reg 1.000000e+04 train accuracy: 0.077510 val accuracy: 0.065000
lr 1.000000e-04 reg 2.500000e+04 train accuracy: 0.082653 val accuracy: 0.075000
lr 1.000000e-04 reg 5.000000e+04 train accuracy: 0.077082 val accuracy: 0.080000
lr 1.000000e-03 reg 1.000000e+03 train accuracy: 0.067449 val accuracy: 0.072000
lr 1.000000e-03 reg 1.000000e+04 train accuracy: 0.082265 val accuracy: 0.082000
lr 1.000000e-03 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.360000
```

[30]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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

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

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



[40]:
```
# Evaluate the best softmax on test set

# softmax = Softmax()
# softmax.train(X_train, y_train, now_lr, reg,num_iters=200, verbose=True) #
  ↪100?
best_softmax = Softmax()
best_softmax.train(X_train,y_train,best_lr,best_reg,num_iters=500,verbose=True)
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('Softmax classifier on raw pixels final test set accuracy: %f' %
  ↪test_accuracy)
```

```
iteration 0 / 500: loss 315.545795
iteration 100 / 500: loss 7.411083
iteration 200 / 500: loss 2.063888
iteration 300 / 500: loss 1.915121
```

```
iteration 400 / 500: loss 1.925587
Softmax classifier on raw pixels final test set accuracy: 0.363000
```

[41]:
```
# Save best softmax model
best_softmax.save("best_softmax.npy")
```

```
best_softmax.npy saved.
```

[42]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
 ↪may
# or may not be nice to look at.
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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

Inline question 3

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

YourAnswer : *fill this in*

Inline Question 4 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would change the softmax loss, but leave the SVM loss unchanged.

YourAnswer :

YourExplanation :

[ ]:

two_layer_net

July 16, 2025

1 Fully-Connected Neural Nets

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

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

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

[2]:
```
# As usual, a bit of setup
from __future__ import print_function
```

```
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

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

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

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

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

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

2 Affine layer: forward

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

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

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

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

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

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

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

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

```
Testing affine_forward function:
difference:  9.769848888397517e-10
```

3 Affine layer: backward

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

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

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

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)
print(dw_num,dw)
# 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))
```

```
[[-2.37342917 -0.51198268  0.31810037  2.90403428  1.06935402]
 [-3.51268592 -2.11141032 -3.63676781 -1.97058092 -2.23413933]
 [ 5.08663365  1.49488732  3.34009108  5.93662486 -2.15844283]
 [-0.69809993 -0.20996862 -2.34841896  2.76050051 -3.16921717]
 [ 2.22793491  2.34320739 -4.92577398  2.06883897 -3.34916043]
 [-0.77798671 -1.29867108 -3.14496814  0.15478615 -3.09581511]] [[-2.37342917
-0.51198268  0.31810037  2.90403428  1.06935402]
 [-3.51268592 -2.11141032 -3.63676781 -1.97058092 -2.23413933]
 [ 5.08663365  1.49488732  3.34009108  5.93662486 -2.15844283]
 [-0.69809993 -0.20996862 -2.34841896  2.76050051 -3.16921717]
 [ 2.22793491  2.34320739 -4.92577398  2.06883897 -3.34916043]
 [-0.77798671 -1.29867108 -3.14496814  0.15478615 -3.09581511]]
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

4 ReLU activation: forward

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

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

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

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

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

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

5 ReLU activation: backward

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

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

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

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

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

5.1 Inline Question 1:

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

YourAnswer : Sigmoid, ReLU.

1. When the input of ReLU is negative, the gradient is 0.
2. When the input of Sigmoid is tend to infinity, the gradient is 0.

6 "Sandwich" layers

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

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

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

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

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

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

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

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

7 Loss layers: Softmax

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

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

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


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

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

```
Testing softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-09
```

8 Two-layer network

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

cell below to test your implementation.

[10]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

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

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

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

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

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

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

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.21e-10
b1 relative error: 8.37e-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: 7.76e-10
```

9 Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

[12]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None
################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
solver = Solver(model, data,
                optim_config={
                  'learning_rate': 0.001,
                })
solver.train()
print(solver.check_accuracy(data['X_test'],data['y_test']))
```

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

```
(Iteration 1 / 4900) loss: 2.302624
{'W1': {'learning_rate': 0.001}, 'b1': {'learning_rate': 0.001}, 'W2':
{'learning_rate': 0.001}, 'b2': {'learning_rate': 0.001}}
{'W1': {'learning_rate': 0.001}, 'b1': {'learning_rate': 0.001}, 'W2':
{'learning_rate': 0.001}, 'b2': {'learning_rate': 0.001}}
(Epoch 0 / 10) train acc: 0.099000; val_acc: 0.102000
(Iteration 11 / 4900) loss: 2.271058
(Iteration 21 / 4900) loss: 2.202720
(Iteration 31 / 4900) loss: 2.098528
(Iteration 41 / 4900) loss: 2.039436
(Iteration 51 / 4900) loss: 1.962531
(Iteration 61 / 4900) loss: 1.994254
(Iteration 71 / 4900) loss: 1.965868
(Iteration 81 / 4900) loss: 1.923882
(Iteration 91 / 4900) loss: 1.787717
(Iteration 101 / 4900) loss: 1.885903
(Iteration 111 / 4900) loss: 1.806981
(Iteration 121 / 4900) loss: 1.851462
(Iteration 131 / 4900) loss: 1.662943
(Iteration 141 / 4900) loss: 1.708006
(Iteration 151 / 4900) loss: 1.917812
(Iteration 161 / 4900) loss: 1.492599
(Iteration 171 / 4900) loss: 1.649813
(Iteration 181 / 4900) loss: 1.699736
(Iteration 191 / 4900) loss: 1.593210
(Iteration 201 / 4900) loss: 1.532704
(Iteration 211 / 4900) loss: 1.804857
(Iteration 221 / 4900) loss: 1.945506
(Iteration 231 / 4900) loss: 1.658519
(Iteration 241 / 4900) loss: 1.750324
(Iteration 251 / 4900) loss: 1.732358
(Iteration 261 / 4900) loss: 1.510227
(Iteration 271 / 4900) loss: 1.663559
(Iteration 281 / 4900) loss: 1.683118
(Iteration 291 / 4900) loss: 1.743562
(Iteration 301 / 4900) loss: 1.783957
(Iteration 311 / 4900) loss: 1.769693
(Iteration 321 / 4900) loss: 1.799049
(Iteration 331 / 4900) loss: 1.382959
(Iteration 341 / 4900) loss: 1.566710
(Iteration 351 / 4900) loss: 1.644926
(Iteration 361 / 4900) loss: 1.707114
(Iteration 371 / 4900) loss: 1.669690
```

```
(Iteration 381 / 4900) loss: 1.509890
(Iteration 391 / 4900) loss: 1.701077
(Iteration 401 / 4900) loss: 1.449410
(Iteration 411 / 4900) loss: 1.531795
(Iteration 421 / 4900) loss: 1.739107
(Iteration 431 / 4900) loss: 1.623026
(Iteration 441 / 4900) loss: 1.741904
(Iteration 451 / 4900) loss: 1.697622
(Iteration 461 / 4900) loss: 1.471479
(Iteration 471 / 4900) loss: 1.518311
(Iteration 481 / 4900) loss: 1.456892
{'W1': {'learning_rate': 0.001}, 'b1': {'learning_rate': 0.001}, 'W2':
{'learning_rate': 0.001}, 'b2': {'learning_rate': 0.001}}
{'W1': {'learning_rate': 0.001}, 'b1': {'learning_rate': 0.001}, 'W2':
{'learning_rate': 0.001}, 'b2': {'learning_rate': 0.001}}
(Epoch 1 / 10) train acc: 0.427000; val_acc: 0.418000
(Iteration 491 / 4900) loss: 1.469874
(Iteration 501 / 4900) loss: 1.762465
(Iteration 511 / 4900) loss: 1.549310
(Iteration 521 / 4900) loss: 1.691392
(Iteration 531 / 4900) loss: 1.459218
(Iteration 541 / 4900) loss: 1.477694
(Iteration 551 / 4900) loss: 1.442633
(Iteration 561 / 4900) loss: 1.461162
(Iteration 571 / 4900) loss: 1.690317
(Iteration 581 / 4900) loss: 1.490683
(Iteration 591 / 4900) loss: 1.464593
(Iteration 601 / 4900) loss: 1.682406
(Iteration 611 / 4900) loss: 1.407564
(Iteration 621 / 4900) loss: 1.586914
(Iteration 631 / 4900) loss: 1.599759
(Iteration 641 / 4900) loss: 1.670674
(Iteration 651 / 4900) loss: 1.392755
(Iteration 661 / 4900) loss: 1.540718
(Iteration 671 / 4900) loss: 1.599092
(Iteration 681 / 4900) loss: 1.441377
(Iteration 691 / 4900) loss: 1.417479
(Iteration 701 / 4900) loss: 1.563928
(Iteration 711 / 4900) loss: 1.407135
(Iteration 721 / 4900) loss: 1.876231
(Iteration 731 / 4900) loss: 1.456168
(Iteration 741 / 4900) loss: 1.580056
(Iteration 751 / 4900) loss: 1.555598
(Iteration 761 / 4900) loss: 1.663925
(Iteration 771 / 4900) loss: 1.558827
(Iteration 781 / 4900) loss: 1.433216
(Iteration 791 / 4900) loss: 1.498131
(Iteration 801 / 4900) loss: 1.680409
```

```
(Iteration 811 / 4900) loss: 1.549790
(Iteration 821 / 4900) loss: 1.550600
(Iteration 831 / 4900) loss: 1.433256
(Iteration 841 / 4900) loss: 1.562667
(Iteration 851 / 4900) loss: 1.720150
(Iteration 861 / 4900) loss: 1.779814
(Iteration 871 / 4900) loss: 1.314455
(Iteration 881 / 4900) loss: 1.533589
(Iteration 891 / 4900) loss: 1.563144
(Iteration 901 / 4900) loss: 1.513259
(Iteration 911 / 4900) loss: 1.521629
(Iteration 921 / 4900) loss: 1.548467
(Iteration 931 / 4900) loss: 1.439754
(Iteration 941 / 4900) loss: 1.511906
(Iteration 951 / 4900) loss: 1.537920
(Iteration 961 / 4900) loss: 1.465809
(Iteration 971 / 4900) loss: 1.350661
{'W1': {'learning_rate': 0.001}, 'b1': {'learning_rate': 0.001}, 'W2':
{'learning_rate': 0.001}, 'b2': {'learning_rate': 0.001}}
{'W1': {'learning_rate': 0.001}, 'b1': {'learning_rate': 0.001}, 'W2':
{'learning_rate': 0.001}, 'b2': {'learning_rate': 0.001}}
(Epoch 2 / 10) train acc: 0.471000; val_acc: 0.455000
(Iteration 981 / 4900) loss: 1.413199
(Iteration 991 / 4900) loss: 1.562318
(Iteration 1001 / 4900) loss: 1.558576
(Iteration 1011 / 4900) loss: 1.630244
(Iteration 1021 / 4900) loss: 1.371085
(Iteration 1031 / 4900) loss: 1.576815
(Iteration 1041 / 4900) loss: 1.267743
(Iteration 1051 / 4900) loss: 1.589393
(Iteration 1061 / 4900) loss: 1.496641
(Iteration 1071 / 4900) loss: 1.542829
(Iteration 1081 / 4900) loss: 1.527522
(Iteration 1091 / 4900) loss: 1.584053
(Iteration 1101 / 4900) loss: 1.439410
(Iteration 1111 / 4900) loss: 1.649835
(Iteration 1121 / 4900) loss: 1.750022
(Iteration 1131 / 4900) loss: 1.605852
(Iteration 1141 / 4900) loss: 1.671687
(Iteration 1151 / 4900) loss: 1.456560
(Iteration 1161 / 4900) loss: 1.450117
(Iteration 1171 / 4900) loss: 1.504764
(Iteration 1181 / 4900) loss: 1.477599
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[12], line 14
```

```
      6␣
 ↪##############################################################################
      7 # TODO: Use a Solver instance to train a TwoLayerNet that achieves abou␣
 ↪36% #
      8 # accuracy on the validation set.                                       ␣
 ↪    #
      9␣
 ↪##############################################################################
     10 solver = Solver(model, data,
     11                  optim_config={
     12                    'learning_rate': 0.001,
     13                  })
---> 14 solver.train()
     15 print(solver.check_accuracy(data['X_test'],data['y_test']))
     17␣
 ↪##############################################################################
     18 #                             END OF YOUR CODE                          ␣
 ↪    #
     19␣
 ↪##############################################################################

File ~/CS231n/assignments/assignment1/cs231n/solver.py:263, in Solver.train(sel
    260 num_iterations = self.num_epochs * iterations_per_epoch
    262 for t in range(num_iterations):
--> 263     self._step()
    265     # Maybe print training loss
    266     if self.verbose and t % self.print_every == 0:

File ~/CS231n/assignments/assignment1/cs231n/solver.py:181, in Solver._step(sel
    178 y_batch = self.y_train[batch_mask]
    180 # Compute loss and gradient
--> 181 loss, grads = self.model.loss(X_batch, y_batch)
    182 self.loss_history.append(loss)
    184 # Perform a parameter update

File ~/CS231n/assignments/assignment1/cs231n/classifiers/fc_net.py:126, in␣
 ↪TwoLayerNet.loss(self, X, y)
    124 d_relu_out,dW2,db2 = affine_backward(daf2_out,af2_cache)
    125 d_af1_out = relu_backward(d_relu_out,relu_cache)
--> 126 dX,dW1,db1 = affine_backward(d_af1_out,af1_cache)
    127 grads = {
    128     'W1': dW1 + self.reg * W1,
    129     'b1': db1.reshape(b1.shape),  #
    130     'W2': dW2 + self.reg * W2,
    131     'b2': db2.reshape(b2.shape)
    132 }
    133␣
 ↪##############################################################################
```

```
    134 #                               END OF YOUR CODE
  ↪  #
    135
 ↪###########################################################################
   (…)    139 # assert grads['W2'].shape == self.params['W2'].shape
    140 # assert grads['b2'].shape == self.params['b2'].shape  #
 ↪ b2 (c_dim,) (1,c_dim)

File ~/CS231n/assignments/assignment1/cs231n/layers.py:65, in
 ↪affine_backward(dout, cache)
     62 dx_flat = dout
     63 # print(dx_flat.shape)
---> 65 dx_flat = np.dot(dx_flat,w.T)  # (N, D)
     66 dx = dx_flat.reshape(x.shape)  # (N, d_1, …, d_k)
     67 r_x = x.reshape(x.shape[0],-1) # (N,D)

KeyboardInterrupt:
```

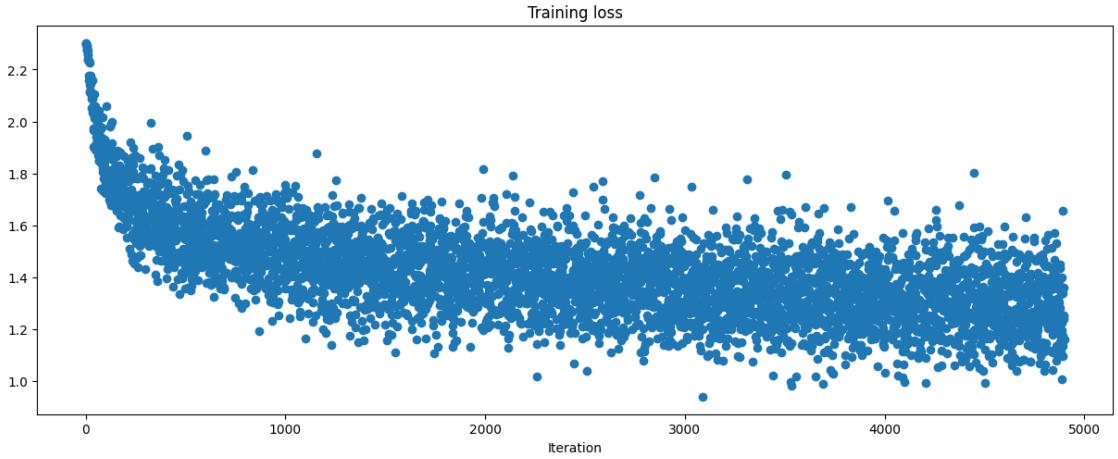
10 Debug the training

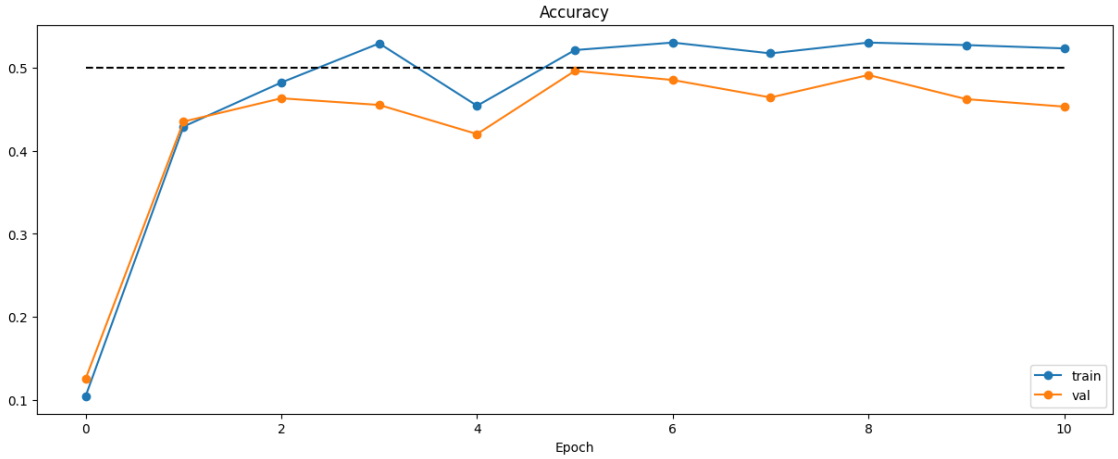
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

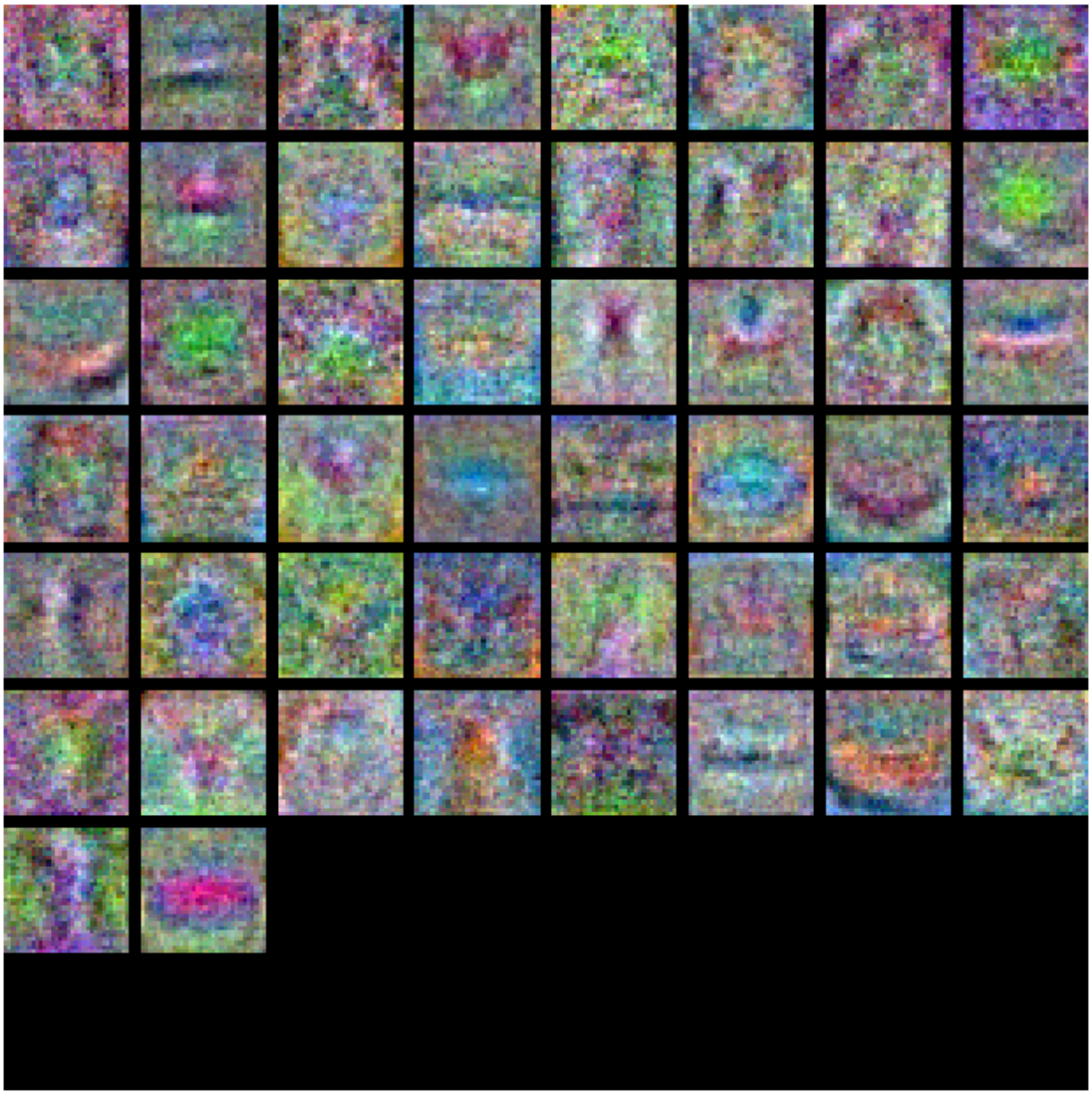
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

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





[ ]:
```python
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



11 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider

tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

[16]:
```
best_model = None
learning_rts = [1e-3,1e-4,1e-5]
reg_strs = [1e-6,1e-2,1,10]

################################################################################
# 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.                       #
################################################################################


input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
best_accuracy = 0
for lr in learning_rts:
    for now_reg in reg_strs:
        model = TwoLayerNet(input_size, hidden_size, num_classes,reg = now_reg)
        solver = Solver(model, data,
                        optim_config={
```

```
                            'learning_rate': lr,
                            },num_epochs=10, batch_size=200,
                    print_every=100)
        solver.train()
        now_acc = solver.check_accuracy(data['X_test'],data['y_test'])
        if now_acc > best_accuracy:
            best_accuracy = now_acc
            best_model = solver
        print(f"lr = %e,reg = %e,acc = %e"%(lr,now_reg,now_acc))

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

```
(Iteration 1 / 2450) loss: 2.303195
(Epoch 0 / 10) train acc: 0.122000; val_acc: 0.112000
(Iteration 101 / 2450) loss: 1.834802
(Iteration 201 / 2450) loss: 1.704401
(Epoch 1 / 10) train acc: 0.413000; val_acc: 0.439000
(Iteration 301 / 2450) loss: 1.716791
(Iteration 401 / 2450) loss: 1.569085
(Epoch 2 / 10) train acc: 0.490000; val_acc: 0.462000
(Iteration 501 / 2450) loss: 1.563251
(Iteration 601 / 2450) loss: 1.516801
(Iteration 701 / 2450) loss: 1.568628
(Epoch 3 / 10) train acc: 0.485000; val_acc: 0.483000
(Iteration 801 / 2450) loss: 1.466507
(Iteration 901 / 2450) loss: 1.475845
(Epoch 4 / 10) train acc: 0.518000; val_acc: 0.480000
(Iteration 1001 / 2450) loss: 1.329374
(Iteration 1101 / 2450) loss: 1.575331
(Iteration 1201 / 2450) loss: 1.356001
(Epoch 5 / 10) train acc: 0.525000; val_acc: 0.473000
(Iteration 1301 / 2450) loss: 1.500471
(Iteration 1401 / 2450) loss: 1.389508
(Epoch 6 / 10) train acc: 0.538000; val_acc: 0.493000
(Iteration 1501 / 2450) loss: 1.382169
(Iteration 1601 / 2450) loss: 1.433500
(Iteration 1701 / 2450) loss: 1.242976
(Epoch 7 / 10) train acc: 0.528000; val_acc: 0.482000
(Iteration 1801 / 2450) loss: 1.372529
(Iteration 1901 / 2450) loss: 1.520136
(Epoch 8 / 10) train acc: 0.538000; val_acc: 0.472000
(Iteration 2001 / 2450) loss: 1.272471
(Iteration 2101 / 2450) loss: 1.274521
(Iteration 2201 / 2450) loss: 1.282659
(Epoch 9 / 10) train acc: 0.526000; val_acc: 0.477000
(Iteration 2301 / 2450) loss: 1.250020
```

```
(Iteration 2401 / 2450) loss: 1.343105
(Epoch 10 / 10) train acc: 0.542000; val_acc: 0.496000
lr = 1.000000e-03,reg = 1.000000e-06,acc = 4.860000e-01
(Iteration 1 / 2450) loss: 2.302176
(Epoch 0 / 10) train acc: 0.138000; val_acc: 0.162000
(Iteration 101 / 2450) loss: 1.804500
(Iteration 201 / 2450) loss: 1.626499
(Epoch 1 / 10) train acc: 0.416000; val_acc: 0.435000
(Iteration 301 / 2450) loss: 1.744984
(Iteration 401 / 2450) loss: 1.672065
(Epoch 2 / 10) train acc: 0.456000; val_acc: 0.432000
(Iteration 501 / 2450) loss: 1.456284
(Iteration 601 / 2450) loss: 1.436688
(Iteration 701 / 2450) loss: 1.587536
(Epoch 3 / 10) train acc: 0.500000; val_acc: 0.453000
(Iteration 801 / 2450) loss: 1.460556
(Iteration 901 / 2450) loss: 1.431577
(Epoch 4 / 10) train acc: 0.491000; val_acc: 0.481000
(Iteration 1001 / 2450) loss: 1.590855
(Iteration 1101 / 2450) loss: 1.357225
(Iteration 1201 / 2450) loss: 1.433052
(Epoch 5 / 10) train acc: 0.509000; val_acc: 0.478000
(Iteration 1301 / 2450) loss: 1.318919
(Iteration 1401 / 2450) loss: 1.406908
(Epoch 6 / 10) train acc: 0.526000; val_acc: 0.476000
(Iteration 1501 / 2450) loss: 1.284485
(Iteration 1601 / 2450) loss: 1.459264
(Iteration 1701 / 2450) loss: 1.386863
(Epoch 7 / 10) train acc: 0.549000; val_acc: 0.482000
(Iteration 1801 / 2450) loss: 1.330014
(Iteration 1901 / 2450) loss: 1.208733
(Epoch 8 / 10) train acc: 0.519000; val_acc: 0.492000
(Iteration 2001 / 2450) loss: 1.413539
(Iteration 2101 / 2450) loss: 1.356045
(Iteration 2201 / 2450) loss: 1.281180
(Epoch 9 / 10) train acc: 0.484000; val_acc: 0.493000
(Iteration 2301 / 2450) loss: 1.335480
(Iteration 2401 / 2450) loss: 1.338368
(Epoch 10 / 10) train acc: 0.557000; val_acc: 0.487000
lr = 1.000000e-03,reg = 1.000000e-02,acc = 4.690000e-01
(Iteration 1 / 2450) loss: 2.374333
(Epoch 0 / 10) train acc: 0.113000; val_acc: 0.148000
(Iteration 101 / 2450) loss: 1.862188
(Iteration 201 / 2450) loss: 1.743175
(Epoch 1 / 10) train acc: 0.413000; val_acc: 0.403000
(Iteration 301 / 2450) loss: 1.640152
(Iteration 401 / 2450) loss: 1.716124
(Epoch 2 / 10) train acc: 0.472000; val_acc: 0.445000
```

```
(Iteration 501 / 2450) loss: 1.710246
(Iteration 601 / 2450) loss: 1.645831
(Iteration 701 / 2450) loss: 1.627317
(Epoch 3 / 10) train acc: 0.462000; val_acc: 0.456000
(Iteration 801 / 2450) loss: 1.562125
(Iteration 901 / 2450) loss: 1.598720
(Epoch 4 / 10) train acc: 0.518000; val_acc: 0.461000
(Iteration 1001 / 2450) loss: 1.564657
(Iteration 1101 / 2450) loss: 1.454142
(Iteration 1201 / 2450) loss: 1.580542
(Epoch 5 / 10) train acc: 0.506000; val_acc: 0.482000
(Iteration 1301 / 2450) loss: 1.581804
(Iteration 1401 / 2450) loss: 1.679067
(Epoch 6 / 10) train acc: 0.512000; val_acc: 0.471000
(Iteration 1501 / 2450) loss: 1.601373
(Iteration 1601 / 2450) loss: 1.517058
(Iteration 1701 / 2450) loss: 1.494250
(Epoch 7 / 10) train acc: 0.520000; val_acc: 0.500000
(Iteration 1801 / 2450) loss: 1.408536
(Iteration 1901 / 2450) loss: 1.540511
(Epoch 8 / 10) train acc: 0.484000; val_acc: 0.462000
(Iteration 2001 / 2450) loss: 1.399968
(Iteration 2101 / 2450) loss: 1.477878
(Iteration 2201 / 2450) loss: 1.664989
(Epoch 9 / 10) train acc: 0.520000; val_acc: 0.487000
(Iteration 2301 / 2450) loss: 1.339551
(Iteration 2401 / 2450) loss: 1.493107
(Epoch 10 / 10) train acc: 0.493000; val_acc: 0.473000
lr = 1.000000e-03,reg = 1.000000e+00,acc = 4.830000e-01
(Iteration 1 / 2450) loss: 3.071646
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.127000
(Iteration 101 / 2450) loss: 2.084085
(Iteration 201 / 2450) loss: 2.039702
(Epoch 1 / 10) train acc: 0.361000; val_acc: 0.363000
(Iteration 301 / 2450) loss: 1.989039
(Iteration 401 / 2450) loss: 1.953050
(Epoch 2 / 10) train acc: 0.389000; val_acc: 0.385000
(Iteration 501 / 2450) loss: 1.968948
(Iteration 601 / 2450) loss: 2.072793
(Iteration 701 / 2450) loss: 1.957734
(Epoch 3 / 10) train acc: 0.377000; val_acc: 0.401000
(Iteration 801 / 2450) loss: 1.884304
(Iteration 901 / 2450) loss: 1.970271
(Epoch 4 / 10) train acc: 0.373000; val_acc: 0.414000
(Iteration 1001 / 2450) loss: 1.984831
(Iteration 1101 / 2450) loss: 1.934399
(Iteration 1201 / 2450) loss: 1.906839
(Epoch 5 / 10) train acc: 0.382000; val_acc: 0.408000
```

```
(Iteration 1301 / 2450) loss: 1.906096
(Iteration 1401 / 2450) loss: 1.988896
(Epoch 6 / 10) train acc: 0.401000; val_acc: 0.402000
(Iteration 1501 / 2450) loss: 1.992758
(Iteration 1601 / 2450) loss: 2.016604
(Iteration 1701 / 2450) loss: 1.918506
(Epoch 7 / 10) train acc: 0.407000; val_acc: 0.397000
(Iteration 1801 / 2450) loss: 1.965769
(Iteration 1901 / 2450) loss: 1.969705
(Epoch 8 / 10) train acc: 0.396000; val_acc: 0.408000
(Iteration 2001 / 2450) loss: 1.931805
(Iteration 2101 / 2450) loss: 2.049273
(Iteration 2201 / 2450) loss: 1.972348
(Epoch 9 / 10) train acc: 0.382000; val_acc: 0.396000
(Iteration 2301 / 2450) loss: 2.056273
(Iteration 2401 / 2450) loss: 1.920175
(Epoch 10 / 10) train acc: 0.408000; val_acc: 0.400000
lr = 1.000000e-03,reg = 1.000000e+01,acc = 4.050000e-01
(Iteration 1 / 2450) loss: 2.307071
(Epoch 0 / 10) train acc: 0.073000; val_acc: 0.084000
(Iteration 101 / 2450) loss: 2.273883
(Iteration 201 / 2450) loss: 2.154511
(Epoch 1 / 10) train acc: 0.239000; val_acc: 0.249000
(Iteration 301 / 2450) loss: 2.104390
(Iteration 401 / 2450) loss: 2.031567
(Epoch 2 / 10) train acc: 0.319000; val_acc: 0.302000
(Iteration 501 / 2450) loss: 1.938396
(Iteration 601 / 2450) loss: 1.915456
(Iteration 701 / 2450) loss: 1.824716
(Epoch 3 / 10) train acc: 0.334000; val_acc: 0.333000
(Iteration 801 / 2450) loss: 1.840958
(Iteration 901 / 2450) loss: 1.859911
(Epoch 4 / 10) train acc: 0.372000; val_acc: 0.361000
(Iteration 1001 / 2450) loss: 1.737307
(Iteration 1101 / 2450) loss: 1.776838
(Iteration 1201 / 2450) loss: 1.806264
(Epoch 5 / 10) train acc: 0.364000; val_acc: 0.388000
(Iteration 1301 / 2450) loss: 1.840936
(Iteration 1401 / 2450) loss: 1.727908
(Epoch 6 / 10) train acc: 0.391000; val_acc: 0.402000
(Iteration 1501 / 2450) loss: 1.738299
(Iteration 1601 / 2450) loss: 1.724917
(Iteration 1701 / 2450) loss: 1.622650
(Epoch 7 / 10) train acc: 0.415000; val_acc: 0.405000
(Iteration 1801 / 2450) loss: 1.611731
(Iteration 1901 / 2450) loss: 1.775696
(Epoch 8 / 10) train acc: 0.427000; val_acc: 0.421000
(Iteration 2001 / 2450) loss: 1.675246
```

```
(Iteration 2101 / 2450) loss: 1.745862
(Iteration 2201 / 2450) loss: 1.611891
(Epoch 9 / 10) train acc: 0.470000; val_acc: 0.431000
(Iteration 2301 / 2450) loss: 1.629254
(Iteration 2401 / 2450) loss: 1.623962
(Epoch 10 / 10) train acc: 0.426000; val_acc: 0.434000
lr = 1.000000e-04,reg = 1.000000e-06,acc = 4.230000e-01
(Iteration 1 / 2450) loss: 2.304444
(Epoch 0 / 10) train acc: 0.081000; val_acc: 0.110000
(Iteration 101 / 2450) loss: 2.264238
(Iteration 201 / 2450) loss: 2.144685
(Epoch 1 / 10) train acc: 0.235000; val_acc: 0.239000
(Iteration 301 / 2450) loss: 2.118426
(Iteration 401 / 2450) loss: 2.090098
(Epoch 2 / 10) train acc: 0.310000; val_acc: 0.305000
(Iteration 501 / 2450) loss: 2.033342
(Iteration 601 / 2450) loss: 1.882461
(Iteration 701 / 2450) loss: 1.883810
(Epoch 3 / 10) train acc: 0.345000; val_acc: 0.334000
(Iteration 801 / 2450) loss: 1.865781
(Iteration 901 / 2450) loss: 1.898236
(Epoch 4 / 10) train acc: 0.359000; val_acc: 0.353000
(Iteration 1001 / 2450) loss: 1.829338
(Iteration 1101 / 2450) loss: 1.866998
(Iteration 1201 / 2450) loss: 1.692500
(Epoch 5 / 10) train acc: 0.385000; val_acc: 0.378000
(Iteration 1301 / 2450) loss: 1.601936
(Iteration 1401 / 2450) loss: 1.794305
(Epoch 6 / 10) train acc: 0.378000; val_acc: 0.395000
(Iteration 1501 / 2450) loss: 1.655469
(Iteration 1601 / 2450) loss: 1.672846
(Iteration 1701 / 2450) loss: 1.702477
(Epoch 7 / 10) train acc: 0.408000; val_acc: 0.409000
(Iteration 1801 / 2450) loss: 1.672893
(Iteration 1901 / 2450) loss: 1.758480
(Epoch 8 / 10) train acc: 0.425000; val_acc: 0.406000
(Iteration 2001 / 2450) loss: 1.593906
(Iteration 2101 / 2450) loss: 1.699202
(Iteration 2201 / 2450) loss: 1.577385
(Epoch 9 / 10) train acc: 0.441000; val_acc: 0.417000
(Iteration 2301 / 2450) loss: 1.506607
(Iteration 2401 / 2450) loss: 1.611858
(Epoch 10 / 10) train acc: 0.436000; val_acc: 0.427000
lr = 1.000000e-04,reg = 1.000000e-02,acc = 4.230000e-01
(Iteration 1 / 2450) loss: 2.378874
(Epoch 0 / 10) train acc: 0.111000; val_acc: 0.118000
(Iteration 101 / 2450) loss: 2.328113
(Iteration 201 / 2450) loss: 2.219742
```

```
(Epoch 1 / 10) train acc: 0.230000; val_acc: 0.250000
(Iteration 301 / 2450) loss: 2.196099
(Iteration 401 / 2450) loss: 2.144066
(Epoch 2 / 10) train acc: 0.268000; val_acc: 0.305000
(Iteration 501 / 2450) loss: 1.989560
(Iteration 601 / 2450) loss: 1.949770
(Iteration 701 / 2450) loss: 1.890879
(Epoch 3 / 10) train acc: 0.340000; val_acc: 0.332000
(Iteration 801 / 2450) loss: 1.987181
(Iteration 901 / 2450) loss: 1.978458
(Epoch 4 / 10) train acc: 0.357000; val_acc: 0.364000
(Iteration 1001 / 2450) loss: 1.991269
(Iteration 1101 / 2450) loss: 2.001872
(Iteration 1201 / 2450) loss: 1.773837
(Epoch 5 / 10) train acc: 0.385000; val_acc: 0.375000
(Iteration 1301 / 2450) loss: 1.812293
(Iteration 1401 / 2450) loss: 1.789782
(Epoch 6 / 10) train acc: 0.383000; val_acc: 0.396000
(Iteration 1501 / 2450) loss: 1.895408
(Iteration 1601 / 2450) loss: 1.912321
(Iteration 1701 / 2450) loss: 1.752303
(Epoch 7 / 10) train acc: 0.407000; val_acc: 0.399000
(Iteration 1801 / 2450) loss: 1.744422
(Iteration 1901 / 2450) loss: 1.803399
(Epoch 8 / 10) train acc: 0.405000; val_acc: 0.418000
(Iteration 2001 / 2450) loss: 1.711733
(Iteration 2101 / 2450) loss: 1.802777
(Iteration 2201 / 2450) loss: 1.650279
(Epoch 9 / 10) train acc: 0.423000; val_acc: 0.427000
(Iteration 2301 / 2450) loss: 1.784874
(Iteration 2401 / 2450) loss: 1.675419
(Epoch 10 / 10) train acc: 0.433000; val_acc: 0.439000
lr = 1.000000e-04,reg = 1.000000e+00,acc = 4.150000e-01
(Iteration 1 / 2450) loss: 3.073207
(Epoch 0 / 10) train acc: 0.077000; val_acc: 0.072000
(Iteration 101 / 2450) loss: 2.903132
(Iteration 201 / 2450) loss: 2.735337
(Epoch 1 / 10) train acc: 0.248000; val_acc: 0.249000
(Iteration 301 / 2450) loss: 2.531403
(Iteration 401 / 2450) loss: 2.425775
(Epoch 2 / 10) train acc: 0.245000; val_acc: 0.280000
(Iteration 501 / 2450) loss: 2.312779
(Iteration 601 / 2450) loss: 2.376827
(Iteration 701 / 2450) loss: 2.166003
(Epoch 3 / 10) train acc: 0.287000; val_acc: 0.308000
(Iteration 801 / 2450) loss: 2.197553
(Iteration 901 / 2450) loss: 2.109854
(Epoch 4 / 10) train acc: 0.336000; val_acc: 0.325000
```

```
(Iteration 1001 / 2450) loss: 1.970633
(Iteration 1101 / 2450) loss: 2.040716
(Iteration 1201 / 2450) loss: 2.072250
(Epoch 5 / 10) train acc: 0.374000; val_acc: 0.345000
(Iteration 1301 / 2450) loss: 2.047389
(Iteration 1401 / 2450) loss: 2.048227
(Epoch 6 / 10) train acc: 0.384000; val_acc: 0.359000
(Iteration 1501 / 2450) loss: 1.943204
(Iteration 1601 / 2450) loss: 1.987733
(Iteration 1701 / 2450) loss: 1.922116
(Epoch 7 / 10) train acc: 0.384000; val_acc: 0.363000
(Iteration 1801 / 2450) loss: 1.943604
(Iteration 1901 / 2450) loss: 2.001169
(Epoch 8 / 10) train acc: 0.363000; val_acc: 0.377000
(Iteration 2001 / 2450) loss: 1.949120
(Iteration 2101 / 2450) loss: 1.900331
(Iteration 2201 / 2450) loss: 1.968607
(Epoch 9 / 10) train acc: 0.365000; val_acc: 0.374000
(Iteration 2301 / 2450) loss: 1.911687
(Iteration 2401 / 2450) loss: 1.976531
(Epoch 10 / 10) train acc: 0.394000; val_acc: 0.379000
lr = 1.000000e-04,reg = 1.000000e+01,acc = 3.820000e-01
(Iteration 1 / 2450) loss: 2.304847
(Epoch 0 / 10) train acc: 0.095000; val_acc: 0.089000
(Iteration 101 / 2450) loss: 2.300739
(Iteration 201 / 2450) loss: 2.299241
(Epoch 1 / 10) train acc: 0.158000; val_acc: 0.166000
(Iteration 301 / 2450) loss: 2.297396
(Iteration 401 / 2450) loss: 2.294631
(Epoch 2 / 10) train acc: 0.193000; val_acc: 0.196000
(Iteration 501 / 2450) loss: 2.292558
(Iteration 601 / 2450) loss: 2.288893
(Iteration 701 / 2450) loss: 2.285044
(Epoch 3 / 10) train acc: 0.213000; val_acc: 0.227000
(Iteration 801 / 2450) loss: 2.279121
(Iteration 901 / 2450) loss: 2.275981
(Epoch 4 / 10) train acc: 0.229000; val_acc: 0.235000
(Iteration 1001 / 2450) loss: 2.260945
(Iteration 1101 / 2450) loss: 2.252053
(Iteration 1201 / 2450) loss: 2.254063
(Epoch 5 / 10) train acc: 0.213000; val_acc: 0.233000
(Iteration 1301 / 2450) loss: 2.221170
(Iteration 1401 / 2450) loss: 2.248338
(Epoch 6 / 10) train acc: 0.212000; val_acc: 0.230000
(Iteration 1501 / 2450) loss: 2.195452
(Iteration 1601 / 2450) loss: 2.206309
(Iteration 1701 / 2450) loss: 2.191668
(Epoch 7 / 10) train acc: 0.213000; val_acc: 0.232000
```

```
(Iteration 1801 / 2450) loss: 2.167974
(Iteration 1901 / 2450) loss: 2.165943
(Epoch 8 / 10) train acc: 0.225000; val_acc: 0.241000
(Iteration 2001 / 2450) loss: 2.136053
(Iteration 2101 / 2450) loss: 2.112513
(Iteration 2201 / 2450) loss: 2.121696
(Epoch 9 / 10) train acc: 0.241000; val_acc: 0.245000
(Iteration 2301 / 2450) loss: 2.150506
(Iteration 2401 / 2450) loss: 2.117900
(Epoch 10 / 10) train acc: 0.245000; val_acc: 0.255000
lr = 1.000000e-05,reg = 1.000000e-06,acc = 2.590000e-01
(Iteration 1 / 2450) loss: 2.304574
(Epoch 0 / 10) train acc: 0.091000; val_acc: 0.093000
(Iteration 101 / 2450) loss: 2.301727
(Iteration 201 / 2450) loss: 2.298840
(Epoch 1 / 10) train acc: 0.155000; val_acc: 0.152000
(Iteration 301 / 2450) loss: 2.293302
(Iteration 401 / 2450) loss: 2.291490
(Epoch 2 / 10) train acc: 0.185000; val_acc: 0.197000
(Iteration 501 / 2450) loss: 2.284704
(Iteration 601 / 2450) loss: 2.286518
(Iteration 701 / 2450) loss: 2.277359
(Epoch 3 / 10) train acc: 0.252000; val_acc: 0.231000
(Iteration 801 / 2450) loss: 2.280461
(Iteration 901 / 2450) loss: 2.264977
(Epoch 4 / 10) train acc: 0.204000; val_acc: 0.232000
(Iteration 1001 / 2450) loss: 2.252543
(Iteration 1101 / 2450) loss: 2.251420
(Iteration 1201 / 2450) loss: 2.223550
(Epoch 5 / 10) train acc: 0.208000; val_acc: 0.241000
(Iteration 1301 / 2450) loss: 2.232942
(Iteration 1401 / 2450) loss: 2.203070
(Epoch 6 / 10) train acc: 0.233000; val_acc: 0.238000
(Iteration 1501 / 2450) loss: 2.200720
(Iteration 1601 / 2450) loss: 2.219623
(Iteration 1701 / 2450) loss: 2.188921
(Epoch 7 / 10) train acc: 0.233000; val_acc: 0.242000
(Iteration 1801 / 2450) loss: 2.158837
(Iteration 1901 / 2450) loss: 2.173986
(Epoch 8 / 10) train acc: 0.240000; val_acc: 0.244000
(Iteration 2001 / 2450) loss: 2.111323
(Iteration 2101 / 2450) loss: 2.129476
(Iteration 2201 / 2450) loss: 2.132409
(Epoch 9 / 10) train acc: 0.241000; val_acc: 0.246000
(Iteration 2301 / 2450) loss: 2.140023
(Iteration 2401 / 2450) loss: 2.089117
(Epoch 10 / 10) train acc: 0.251000; val_acc: 0.255000
lr = 1.000000e-05,reg = 1.000000e-02,acc = 2.600000e-01
```

```
(Iteration 1 / 2450) loss: 2.378743
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.079000
(Iteration 101 / 2450) loss: 2.375814
(Iteration 201 / 2450) loss: 2.374435
(Epoch 1 / 10) train acc: 0.145000; val_acc: 0.135000
(Iteration 301 / 2450) loss: 2.375405
(Iteration 401 / 2450) loss: 2.371826
(Epoch 2 / 10) train acc: 0.195000; val_acc: 0.169000
(Iteration 501 / 2450) loss: 2.364446
(Iteration 601 / 2450) loss: 2.358453
(Iteration 701 / 2450) loss: 2.357708
(Epoch 3 / 10) train acc: 0.211000; val_acc: 0.189000
(Iteration 801 / 2450) loss: 2.346564
(Iteration 901 / 2450) loss: 2.340204
(Epoch 4 / 10) train acc: 0.226000; val_acc: 0.214000
(Iteration 1001 / 2450) loss: 2.337856
(Iteration 1101 / 2450) loss: 2.339684
(Iteration 1201 / 2450) loss: 2.328934
(Epoch 5 / 10) train acc: 0.219000; val_acc: 0.229000
(Iteration 1301 / 2450) loss: 2.319641
(Iteration 1401 / 2450) loss: 2.298113
(Epoch 6 / 10) train acc: 0.220000; val_acc: 0.235000
(Iteration 1501 / 2450) loss: 2.281244
(Iteration 1601 / 2450) loss: 2.287744
(Iteration 1701 / 2450) loss: 2.267284
(Epoch 7 / 10) train acc: 0.257000; val_acc: 0.236000
(Iteration 1801 / 2450) loss: 2.212154
(Iteration 1901 / 2450) loss: 2.231424
(Epoch 8 / 10) train acc: 0.237000; val_acc: 0.243000
(Iteration 2001 / 2450) loss: 2.218685
(Iteration 2101 / 2450) loss: 2.218903
(Iteration 2201 / 2450) loss: 2.184645
(Epoch 9 / 10) train acc: 0.244000; val_acc: 0.249000
(Iteration 2301 / 2450) loss: 2.147299
(Iteration 2401 / 2450) loss: 2.193153
(Epoch 10 / 10) train acc: 0.256000; val_acc: 0.257000
lr = 1.000000e-05,reg = 1.000000e+00,acc = 2.720000e-01
(Iteration 1 / 2450) loss: 3.069442
(Epoch 0 / 10) train acc: 0.083000; val_acc: 0.092000
(Iteration 101 / 2450) loss: 3.053072
(Iteration 201 / 2450) loss: 3.036511
(Epoch 1 / 10) train acc: 0.151000; val_acc: 0.166000
(Iteration 301 / 2450) loss: 3.019774
(Iteration 401 / 2450) loss: 3.003158
(Epoch 2 / 10) train acc: 0.190000; val_acc: 0.192000
(Iteration 501 / 2450) loss: 2.987669
(Iteration 601 / 2450) loss: 2.970731
(Iteration 701 / 2450) loss: 2.957129
```

```
(Epoch 3 / 10) train acc: 0.190000; val_acc: 0.203000
(Iteration 801 / 2450) loss: 2.940794
(Iteration 901 / 2450) loss: 2.928979
(Epoch 4 / 10) train acc: 0.225000; val_acc: 0.205000
(Iteration 1001 / 2450) loss: 2.901310
(Iteration 1101 / 2450) loss: 2.883336
(Iteration 1201 / 2450) loss: 2.874717
(Epoch 5 / 10) train acc: 0.217000; val_acc: 0.197000
(Iteration 1301 / 2450) loss: 2.858945
(Iteration 1401 / 2450) loss: 2.825336
(Epoch 6 / 10) train acc: 0.179000; val_acc: 0.194000
(Iteration 1501 / 2450) loss: 2.817570
(Iteration 1601 / 2450) loss: 2.802701
(Iteration 1701 / 2450) loss: 2.771800
(Epoch 7 / 10) train acc: 0.190000; val_acc: 0.198000
(Iteration 1801 / 2450) loss: 2.757142
(Iteration 1901 / 2450) loss: 2.736139
(Epoch 8 / 10) train acc: 0.188000; val_acc: 0.217000
(Iteration 2001 / 2450) loss: 2.738918
(Iteration 2101 / 2450) loss: 2.711804
(Iteration 2201 / 2450) loss: 2.667229
(Epoch 9 / 10) train acc: 0.198000; val_acc: 0.225000
(Iteration 2301 / 2450) loss: 2.676139
(Iteration 2401 / 2450) loss: 2.675291
(Epoch 10 / 10) train acc: 0.207000; val_acc: 0.238000
lr = 1.000000e-05,reg = 1.000000e+01,acc = 2.440000e-01
```

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.

[21]:
```
# y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', best_model.
  ↪check_accuracy(data['X_test'],data['y_test']))
```

```
Validation set accuracy:  0.486
```

[19]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[19], line 1
----> 1 y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
      2 print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
AttributeError: 'Solver' object has no attribute 'loss'
```

[22]:
```
# Save best model
best_model.save("best_two_layer_net.npy")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[22], line 2
      1 # Save best model
----> 2 best_model.save("best_two_layer_net.npy")

AttributeError: 'Solver' object has no attribute 'save'
```

12.1 Inline Question 2:

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

YourAnswer :

YourExplanation :

[ ]:

features

July 16, 2025

1 Image features exercise

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

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

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

[3]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt
import scipy

%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
```

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

1.1 Load data

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

[4]:
```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
```

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

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

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

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

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

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

1.2 Extract Features

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

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

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

[5]:
```
from cs231n.features import *

# num_color_bins = 10 # Number of bins in the color histogram
```

```
num_color_bins = 25 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

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

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

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
```

```
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train Softmax classifier on features

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

[15]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import Softmax

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

results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the Softmax; save#
```

```
# the best trained classifer in best_softmax. If you carefully tune the model, #
# you should be able to get accuracy of above 0.42 on the validation set.      #
################################################################################

for now_lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train_feats, y_train, now_lr, reg,num_iters=1000,␣
 ↪verbose=True) #   100?
        y_train_pred = softmax.predict(X_train_feats)
        tr_acc = np.mean(y_train == y_train_pred)
        y_val_pred = softmax.predict(X_val_feats)
        val_acc = np.mean(y_val == y_val_pred)
        results[(now_lr,reg)] = (tr_acc,val_acc)
        if val_acc > best_val:
            best_val = val_acc
            best_softmax = softmax
            best_lr,best_reg = now_lr,reg


# 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 / 1000: loss 831.113842
iteration 100 / 1000: loss 2.302585
iteration 200 / 1000: loss 2.302585
iteration 300 / 1000: loss 2.302585
iteration 400 / 1000: loss 2.302585
iteration 500 / 1000: loss 2.302585
iteration 600 / 1000: loss 2.302585
iteration 700 / 1000: loss 2.302585
iteration 800 / 1000: loss 2.302585
iteration 900 / 1000: loss 2.302585
iteration 0 / 1000: loss 8106.716898
iteration 100 / 1000: loss 2.302585
iteration 200 / 1000: loss 2.302585
iteration 300 / 1000: loss 2.302585
iteration 400 / 1000: loss 2.302585
iteration 500 / 1000: loss 2.302585
iteration 600 / 1000: loss 2.302585
iteration 700 / 1000: loss 2.302585
iteration 800 / 1000: loss 2.302585
iteration 900 / 1000: loss 2.302585
```

```
iteration 0 / 1000: loss 846.667608
iteration 100 / 1000: loss 2.302585
iteration 200 / 1000: loss 2.302585
iteration 300 / 1000: loss 2.302585
iteration 400 / 1000: loss 2.302585
iteration 500 / 1000: loss 2.302585
iteration 600 / 1000: loss 2.302585
iteration 700 / 1000: loss 2.302585
iteration 800 / 1000: loss 2.302585
iteration 900 / 1000: loss 2.302585
iteration 0 / 1000: loss 8210.462729
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:81:
RuntimeWarning: divide by zero encountered in log
  cp = -np.log(prob[np.arange(num_train),y])
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:82:
RuntimeWarning: overflow encountered in scalar multiply
  loss = np.sum(cp) / num_train + reg * np.sum(W * W)
/home/doubeecat/CS231n/.venv/lib/python3.12/site-
packages/numpy/_core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:82:
RuntimeWarning: overflow encountered in multiply
  loss = np.sum(cp) / num_train + reg * np.sum(W * W)
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:88:
RuntimeWarning: overflow encountered in multiply
  dW = np.dot(X.T,ds) + 2 * reg * W
/home/doubeecat/CS231n/assignments/assignment1/cs231n/classifiers/softmax.py:76:
RuntimeWarning: invalid value encountered in dot
  scores = X.dot(W)
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.414755 val accuracy: 0.414000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.333429 val accuracy: 0.319000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.313286 val accuracy: 0.305000
lr 1.000000e-06 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved: 0.414000
```

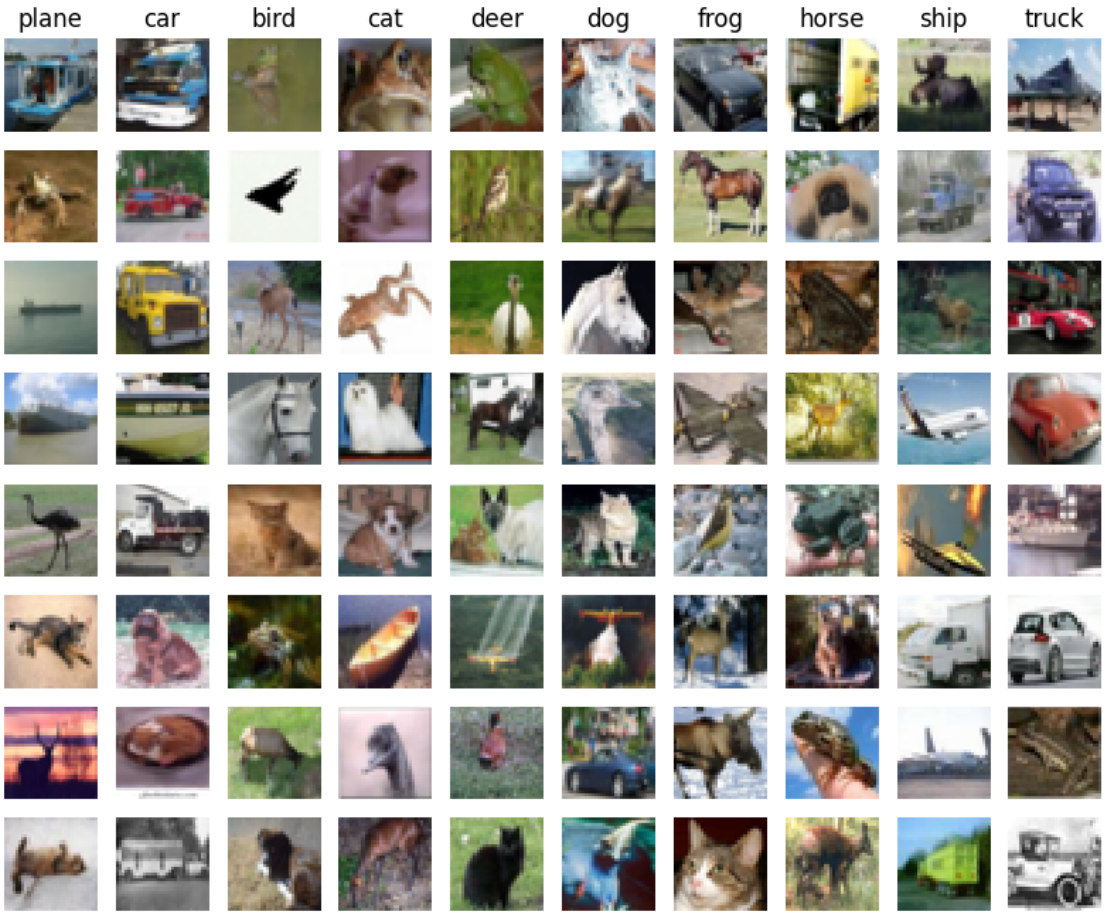
[16]:
```
# Evaluate your trained Softmax on the test set: you should be able to get at␣
 ↪least 0.42
y_test_pred = best_softmax.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.414
```

[17]:
```
# Save best softmax model
best_softmax.save("best_softmax_features.npy")
```

```
best_softmax_features.npy saved.
```

[18]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

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



1.3.1 Inline question 1:

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

YourAnswer : no!

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.

[19]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
```

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

print(X_train_feats.shape)
```

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

[28]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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

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

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None
best_accuracy = 0

################################################################################
# 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.                                              #
################################################################################

best_model = None
learning_rts = [1e-3,1e-2,2.75e-2]
reg_strs = [1e-6,1e-2,1]
for lr in learning_rts:
    for now_reg in reg_strs:
        net = TwoLayerNet(input_dim, hidden_dim, num_classes,reg = now_reg)
        solver = Solver(net, data,
                        optim_config={
                        'learning_rate': lr,
                        },num_epochs=20, batch_size=200,
                    print_every=100)
        solver.train()
        now_acc = solver.best_val_acc
```

```
        if now_acc > best_accuracy:
            best_accuracy = now_acc
            best_net = solver
        print(f"lr = %e,reg = %e,acc = %e"%(lr,now_reg,now_acc))
```

```
(Iteration 1 / 4900) loss: 2.302568
(Epoch 0 / 20) train acc: 0.117000; val_acc: 0.113000
(Iteration 101 / 4900) loss: 2.302570
(Iteration 201 / 4900) loss: 2.302547
(Epoch 1 / 20) train acc: 0.109000; val_acc: 0.112000
(Iteration 301 / 4900) loss: 2.302530
(Iteration 401 / 4900) loss: 2.302458
(Epoch 2 / 20) train acc: 0.173000; val_acc: 0.164000
(Iteration 501 / 4900) loss: 2.302442
(Iteration 601 / 4900) loss: 2.302426
(Iteration 701 / 4900) loss: 2.302426
(Epoch 3 / 20) train acc: 0.164000; val_acc: 0.155000
(Iteration 801 / 4900) loss: 2.302473
(Iteration 901 / 4900) loss: 2.302413
(Epoch 4 / 20) train acc: 0.153000; val_acc: 0.174000
(Iteration 1001 / 4900) loss: 2.302346
(Iteration 1101 / 4900) loss: 2.302340
(Iteration 1201 / 4900) loss: 2.302396
(Epoch 5 / 20) train acc: 0.174000; val_acc: 0.180000
(Iteration 1301 / 4900) loss: 2.302309
(Iteration 1401 / 4900) loss: 2.302331
(Epoch 6 / 20) train acc: 0.151000; val_acc: 0.153000
(Iteration 1501 / 4900) loss: 2.302229
(Iteration 1601 / 4900) loss: 2.302224
(Iteration 1701 / 4900) loss: 2.302183
(Epoch 7 / 20) train acc: 0.190000; val_acc: 0.197000
(Iteration 1801 / 4900) loss: 2.302133
(Iteration 1901 / 4900) loss: 2.302144
(Epoch 8 / 20) train acc: 0.188000; val_acc: 0.192000
(Iteration 2001 / 4900) loss: 2.302146
(Iteration 2101 / 4900) loss: 2.301973
(Iteration 2201 / 4900) loss: 2.302142
(Epoch 9 / 20) train acc: 0.187000; val_acc: 0.182000
(Iteration 2301 / 4900) loss: 2.301917
(Iteration 2401 / 4900) loss: 2.301804
(Epoch 10 / 20) train acc: 0.224000; val_acc: 0.206000
(Iteration 2501 / 4900) loss: 2.301940
(Iteration 2601 / 4900) loss: 2.301687
(Epoch 11 / 20) train acc: 0.231000; val_acc: 0.193000
(Iteration 2701 / 4900) loss: 2.301855
(Iteration 2801 / 4900) loss: 2.301646
(Iteration 2901 / 4900) loss: 2.301542
(Epoch 12 / 20) train acc: 0.222000; val_acc: 0.189000
```

```
(Iteration 3001 / 4900) loss: 2.301655
(Iteration 3101 / 4900) loss: 2.301624
(Epoch 13 / 20) train acc: 0.217000; val_acc: 0.175000
(Iteration 3201 / 4900) loss: 2.301486
(Iteration 3301 / 4900) loss: 2.301455
(Iteration 3401 / 4900) loss: 2.301461
(Epoch 14 / 20) train acc: 0.231000; val_acc: 0.201000
(Iteration 3501 / 4900) loss: 2.301357
(Iteration 3601 / 4900) loss: 2.301396
(Epoch 15 / 20) train acc: 0.228000; val_acc: 0.184000
(Iteration 3701 / 4900) loss: 2.301161
(Iteration 3801 / 4900) loss: 2.301144
(Iteration 3901 / 4900) loss: 2.300905
(Epoch 16 / 20) train acc: 0.205000; val_acc: 0.213000
(Iteration 4001 / 4900) loss: 2.300715
(Iteration 4101 / 4900) loss: 2.300413
(Epoch 17 / 20) train acc: 0.207000; val_acc: 0.200000
(Iteration 4201 / 4900) loss: 2.300358
(Iteration 4301 / 4900) loss: 2.300588
(Iteration 4401 / 4900) loss: 2.300112
(Epoch 18 / 20) train acc: 0.256000; val_acc: 0.205000
(Iteration 4501 / 4900) loss: 2.300050
(Iteration 4601 / 4900) loss: 2.299762
(Epoch 19 / 20) train acc: 0.213000; val_acc: 0.204000
(Iteration 4701 / 4900) loss: 2.299422
(Iteration 4801 / 4900) loss: 2.299226
(Epoch 20 / 20) train acc: 0.242000; val_acc: 0.208000
lr = 1.000000e-03,reg = 1.000000e-06,acc = 2.130000e-01
(Iteration 1 / 4900) loss: 2.303029
(Epoch 0 / 20) train acc: 0.105000; val_acc: 0.097000
(Iteration 101 / 4900) loss: 2.302973
(Iteration 201 / 4900) loss: 2.303039
(Epoch 1 / 20) train acc: 0.100000; val_acc: 0.108000
(Iteration 301 / 4900) loss: 2.302902
(Iteration 401 / 4900) loss: 2.302926
(Epoch 2 / 20) train acc: 0.132000; val_acc: 0.125000
(Iteration 501 / 4900) loss: 2.302941
(Iteration 601 / 4900) loss: 2.302942
(Iteration 701 / 4900) loss: 2.302838
(Epoch 3 / 20) train acc: 0.141000; val_acc: 0.159000
(Iteration 801 / 4900) loss: 2.302875
(Iteration 901 / 4900) loss: 2.302756
(Epoch 4 / 20) train acc: 0.172000; val_acc: 0.164000
(Iteration 1001 / 4900) loss: 2.302865
(Iteration 1101 / 4900) loss: 2.302820
(Iteration 1201 / 4900) loss: 2.302896
(Epoch 5 / 20) train acc: 0.161000; val_acc: 0.153000
(Iteration 1301 / 4900) loss: 2.302704
```

```
(Iteration 1401 / 4900) loss: 2.302707
(Epoch 6 / 20) train acc: 0.171000; val_acc: 0.131000
(Iteration 1501 / 4900) loss: 2.302766
(Iteration 1601 / 4900) loss: 2.302790
(Iteration 1701 / 4900) loss: 2.302584
(Epoch 7 / 20) train acc: 0.152000; val_acc: 0.134000
(Iteration 1801 / 4900) loss: 2.302667
(Iteration 1901 / 4900) loss: 2.302468
(Epoch 8 / 20) train acc: 0.148000; val_acc: 0.127000
(Iteration 2001 / 4900) loss: 2.302551
(Iteration 2101 / 4900) loss: 2.302470
(Iteration 2201 / 4900) loss: 2.302625
(Epoch 9 / 20) train acc: 0.171000; val_acc: 0.130000
(Iteration 2301 / 4900) loss: 2.302613
(Iteration 2401 / 4900) loss: 2.302669
(Epoch 10 / 20) train acc: 0.147000; val_acc: 0.137000
(Iteration 2501 / 4900) loss: 2.302446
(Iteration 2601 / 4900) loss: 2.302333
(Epoch 11 / 20) train acc: 0.187000; val_acc: 0.158000
(Iteration 2701 / 4900) loss: 2.302369
(Iteration 2801 / 4900) loss: 2.302137
(Iteration 2901 / 4900) loss: 2.302260
(Epoch 12 / 20) train acc: 0.177000; val_acc: 0.167000
(Iteration 3001 / 4900) loss: 2.302077
(Iteration 3101 / 4900) loss: 2.302045
(Epoch 13 / 20) train acc: 0.212000; val_acc: 0.171000
(Iteration 3201 / 4900) loss: 2.302056
(Iteration 3301 / 4900) loss: 2.301994
(Iteration 3401 / 4900) loss: 2.301946
(Epoch 14 / 20) train acc: 0.224000; val_acc: 0.189000
(Iteration 3501 / 4900) loss: 2.302129
(Iteration 3601 / 4900) loss: 2.301687
(Epoch 15 / 20) train acc: 0.238000; val_acc: 0.189000
(Iteration 3701 / 4900) loss: 2.301858
(Iteration 3801 / 4900) loss: 2.301539
(Iteration 3901 / 4900) loss: 2.301467
(Epoch 16 / 20) train acc: 0.203000; val_acc: 0.186000
(Iteration 4001 / 4900) loss: 2.301694
(Iteration 4101 / 4900) loss: 2.301398
(Epoch 17 / 20) train acc: 0.259000; val_acc: 0.224000
(Iteration 4201 / 4900) loss: 2.301165
(Iteration 4301 / 4900) loss: 2.301102
(Iteration 4401 / 4900) loss: 2.300973
(Epoch 18 / 20) train acc: 0.279000; val_acc: 0.242000
(Iteration 4501 / 4900) loss: 2.301110
(Iteration 4601 / 4900) loss: 2.300960
(Epoch 19 / 20) train acc: 0.284000; val_acc: 0.249000
(Iteration 4701 / 4900) loss: 2.300543
```

```
(Iteration 4801 / 4900) loss: 2.300155
(Epoch 20 / 20) train acc: 0.313000; val_acc: 0.250000
lr = 1.000000e-03,reg = 1.000000e-02,acc = 2.500000e-01
(Iteration 1 / 4900) loss: 2.347513
(Epoch 0 / 20) train acc: 0.083000; val_acc: 0.095000
(Iteration 101 / 4900) loss: 2.339389
(Iteration 201 / 4900) loss: 2.332664
(Epoch 1 / 20) train acc: 0.093000; val_acc: 0.108000
(Iteration 301 / 4900) loss: 2.327221
(Iteration 401 / 4900) loss: 2.322690
(Epoch 2 / 20) train acc: 0.118000; val_acc: 0.123000
(Iteration 501 / 4900) loss: 2.319138
(Iteration 601 / 4900) loss: 2.316101
(Iteration 701 / 4900) loss: 2.313621
(Epoch 3 / 20) train acc: 0.112000; val_acc: 0.098000
(Iteration 801 / 4900) loss: 2.311667
(Iteration 901 / 4900) loss: 2.309930
(Epoch 4 / 20) train acc: 0.100000; val_acc: 0.098000
(Iteration 1001 / 4900) loss: 2.308724
(Iteration 1101 / 4900) loss: 2.307535
(Iteration 1201 / 4900) loss: 2.306653
(Epoch 5 / 20) train acc: 0.107000; val_acc: 0.100000
(Iteration 1301 / 4900) loss: 2.305895
(Iteration 1401 / 4900) loss: 2.305313
(Epoch 6 / 20) train acc: 0.093000; val_acc: 0.078000
(Iteration 1501 / 4900) loss: 2.304770
(Iteration 1601 / 4900) loss: 2.304478
(Iteration 1701 / 4900) loss: 2.304102
(Epoch 7 / 20) train acc: 0.085000; val_acc: 0.098000
(Iteration 1801 / 4900) loss: 2.303707
(Iteration 1901 / 4900) loss: 2.303644
(Epoch 8 / 20) train acc: 0.104000; val_acc: 0.098000
(Iteration 2001 / 4900) loss: 2.303445
(Iteration 2101 / 4900) loss: 2.303283
(Iteration 2201 / 4900) loss: 2.303163
(Epoch 9 / 20) train acc: 0.104000; val_acc: 0.098000
(Iteration 2301 / 4900) loss: 2.303086
(Iteration 2401 / 4900) loss: 2.302840
(Epoch 10 / 20) train acc: 0.105000; val_acc: 0.078000
(Iteration 2501 / 4900) loss: 2.302949
(Iteration 2601 / 4900) loss: 2.302774
(Epoch 11 / 20) train acc: 0.101000; val_acc: 0.078000
(Iteration 2701 / 4900) loss: 2.302835
(Iteration 2801 / 4900) loss: 2.302693
(Iteration 2901 / 4900) loss: 2.302483
(Epoch 12 / 20) train acc: 0.083000; val_acc: 0.078000
(Iteration 3001 / 4900) loss: 2.302636
(Iteration 3101 / 4900) loss: 2.302747
```

```
(Epoch 13 / 20) train acc: 0.122000; val_acc: 0.078000
(Iteration 3201 / 4900) loss: 2.302725
(Iteration 3301 / 4900) loss: 2.302671
(Iteration 3401 / 4900) loss: 2.302569
(Epoch 14 / 20) train acc: 0.089000; val_acc: 0.078000
(Iteration 3501 / 4900) loss: 2.302645
(Iteration 3601 / 4900) loss: 2.302615
(Epoch 15 / 20) train acc: 0.104000; val_acc: 0.078000
(Iteration 3701 / 4900) loss: 2.302553
(Iteration 3801 / 4900) loss: 2.302594
(Iteration 3901 / 4900) loss: 2.302563
(Epoch 16 / 20) train acc: 0.101000; val_acc: 0.078000
(Iteration 4001 / 4900) loss: 2.302816
(Iteration 4101 / 4900) loss: 2.302699
(Epoch 17 / 20) train acc: 0.079000; val_acc: 0.078000
(Iteration 4201 / 4900) loss: 2.302647
(Iteration 4301 / 4900) loss: 2.302731
(Iteration 4401 / 4900) loss: 2.302714
(Epoch 18 / 20) train acc: 0.098000; val_acc: 0.078000
(Iteration 4501 / 4900) loss: 2.302456
(Iteration 4601 / 4900) loss: 2.302596
(Epoch 19 / 20) train acc: 0.108000; val_acc: 0.078000
(Iteration 4701 / 4900) loss: 2.302534
(Iteration 4801 / 4900) loss: 2.302599
(Epoch 20 / 20) train acc: 0.076000; val_acc: 0.078000
lr = 1.000000e-03,reg = 1.000000e+00,acc = 1.230000e-01
(Iteration 1 / 4900) loss: 2.302621
(Epoch 0 / 20) train acc: 0.083000; val_acc: 0.085000
(Iteration 101 / 4900) loss: 2.302303
(Iteration 201 / 4900) loss: 2.302370
(Epoch 1 / 20) train acc: 0.135000; val_acc: 0.134000
(Iteration 301 / 4900) loss: 2.301596
(Iteration 401 / 4900) loss: 2.300846
(Epoch 2 / 20) train acc: 0.143000; val_acc: 0.128000
(Iteration 501 / 4900) loss: 2.298771
(Iteration 601 / 4900) loss: 2.295586
(Iteration 701 / 4900) loss: 2.289302
(Epoch 3 / 20) train acc: 0.234000; val_acc: 0.195000
(Iteration 801 / 4900) loss: 2.280467
(Iteration 901 / 4900) loss: 2.244998
(Epoch 4 / 20) train acc: 0.244000; val_acc: 0.260000
(Iteration 1001 / 4900) loss: 2.196734
(Iteration 1101 / 4900) loss: 2.184101
(Iteration 1201 / 4900) loss: 2.135753
(Epoch 5 / 20) train acc: 0.277000; val_acc: 0.282000
(Iteration 1301 / 4900) loss: 2.033242
(Iteration 1401 / 4900) loss: 1.963594
(Epoch 6 / 20) train acc: 0.319000; val_acc: 0.306000
```

```
(Iteration 1501 / 4900) loss: 1.979136
(Iteration 1601 / 4900) loss: 1.876996
(Iteration 1701 / 4900) loss: 1.929026
(Epoch 7 / 20) train acc: 0.352000; val_acc: 0.346000
(Iteration 1801 / 4900) loss: 1.783618
(Iteration 1901 / 4900) loss: 1.799748
(Epoch 8 / 20) train acc: 0.384000; val_acc: 0.375000
(Iteration 2001 / 4900) loss: 1.721992
(Iteration 2101 / 4900) loss: 1.714836
(Iteration 2201 / 4900) loss: 1.682679
(Epoch 9 / 20) train acc: 0.416000; val_acc: 0.403000
(Iteration 2301 / 4900) loss: 1.663300
(Iteration 2401 / 4900) loss: 1.650147
(Epoch 10 / 20) train acc: 0.431000; val_acc: 0.418000
(Iteration 2501 / 4900) loss: 1.595632
(Iteration 2601 / 4900) loss: 1.548639
(Epoch 11 / 20) train acc: 0.458000; val_acc: 0.427000
(Iteration 2701 / 4900) loss: 1.369552
(Iteration 2801 / 4900) loss: 1.488848
(Iteration 2901 / 4900) loss: 1.505854
(Epoch 12 / 20) train acc: 0.445000; val_acc: 0.448000
(Iteration 3001 / 4900) loss: 1.552679
(Iteration 3101 / 4900) loss: 1.657576
(Epoch 13 / 20) train acc: 0.467000; val_acc: 0.466000
(Iteration 3201 / 4900) loss: 1.560920
(Iteration 3301 / 4900) loss: 1.405427
(Iteration 3401 / 4900) loss: 1.511940
(Epoch 14 / 20) train acc: 0.475000; val_acc: 0.474000
(Iteration 3501 / 4900) loss: 1.467290
(Iteration 3601 / 4900) loss: 1.492958
(Epoch 15 / 20) train acc: 0.487000; val_acc: 0.487000
(Iteration 3701 / 4900) loss: 1.467519
(Iteration 3801 / 4900) loss: 1.520215
(Iteration 3901 / 4900) loss: 1.497252
(Epoch 16 / 20) train acc: 0.483000; val_acc: 0.495000
(Iteration 4001 / 4900) loss: 1.521552
(Iteration 4101 / 4900) loss: 1.512870
(Epoch 17 / 20) train acc: 0.519000; val_acc: 0.503000
(Iteration 4201 / 4900) loss: 1.402339
(Iteration 4301 / 4900) loss: 1.262279
(Iteration 4401 / 4900) loss: 1.454570
(Epoch 18 / 20) train acc: 0.517000; val_acc: 0.506000
(Iteration 4501 / 4900) loss: 1.376756
(Iteration 4601 / 4900) loss: 1.408529
(Epoch 19 / 20) train acc: 0.501000; val_acc: 0.514000
(Iteration 4701 / 4900) loss: 1.308565
(Iteration 4801 / 4900) loss: 1.338929
(Epoch 20 / 20) train acc: 0.538000; val_acc: 0.510000
```

```
lr = 1.000000e-02,reg = 1.000000e-06,acc = 5.140000e-01
(Iteration 1 / 4900) loss: 2.303058
(Epoch 0 / 20) train acc: 0.098000; val_acc: 0.083000
(Iteration 101 / 4900) loss: 2.302734
(Iteration 201 / 4900) loss: 2.302727
(Epoch 1 / 20) train acc: 0.098000; val_acc: 0.113000
(Iteration 301 / 4900) loss: 2.302184
(Iteration 401 / 4900) loss: 2.301850
(Epoch 2 / 20) train acc: 0.184000; val_acc: 0.182000
(Iteration 501 / 4900) loss: 2.300841
(Iteration 601 / 4900) loss: 2.298655
(Iteration 701 / 4900) loss: 2.292766
(Epoch 3 / 20) train acc: 0.259000; val_acc: 0.267000
(Iteration 801 / 4900) loss: 2.285660
(Iteration 901 / 4900) loss: 2.264695
(Epoch 4 / 20) train acc: 0.276000; val_acc: 0.265000
(Iteration 1001 / 4900) loss: 2.226494
(Iteration 1101 / 4900) loss: 2.204242
(Iteration 1201 / 4900) loss: 2.176164
(Epoch 5 / 20) train acc: 0.261000; val_acc: 0.262000
(Iteration 1301 / 4900) loss: 2.080878
(Iteration 1401 / 4900) loss: 2.063246
(Epoch 6 / 20) train acc: 0.283000; val_acc: 0.298000
(Iteration 1501 / 4900) loss: 1.991875
(Iteration 1601 / 4900) loss: 1.956454
(Iteration 1701 / 4900) loss: 1.912698
(Epoch 7 / 20) train acc: 0.323000; val_acc: 0.335000
(Iteration 1801 / 4900) loss: 1.821414
(Iteration 1901 / 4900) loss: 1.852869
(Epoch 8 / 20) train acc: 0.341000; val_acc: 0.366000
(Iteration 2001 / 4900) loss: 1.793908
(Iteration 2101 / 4900) loss: 1.789246
(Iteration 2201 / 4900) loss: 1.755029
(Epoch 9 / 20) train acc: 0.413000; val_acc: 0.379000
(Iteration 2301 / 4900) loss: 1.721689
(Iteration 2401 / 4900) loss: 1.723471
(Epoch 10 / 20) train acc: 0.416000; val_acc: 0.398000
(Iteration 2501 / 4900) loss: 1.745249
(Iteration 2601 / 4900) loss: 1.748471
(Epoch 11 / 20) train acc: 0.420000; val_acc: 0.412000
(Iteration 2701 / 4900) loss: 1.648204
(Iteration 2801 / 4900) loss: 1.687797
(Iteration 2901 / 4900) loss: 1.648584
(Epoch 12 / 20) train acc: 0.442000; val_acc: 0.428000
(Iteration 3001 / 4900) loss: 1.573198
(Iteration 3101 / 4900) loss: 1.543683
(Epoch 13 / 20) train acc: 0.457000; val_acc: 0.452000
(Iteration 3201 / 4900) loss: 1.595332
```

```
(Iteration 3301 / 4900) loss: 1.642795
(Iteration 3401 / 4900) loss: 1.490602
(Epoch 14 / 20) train acc: 0.448000; val_acc: 0.463000
(Iteration 3501 / 4900) loss: 1.460074
(Iteration 3601 / 4900) loss: 1.483512
(Epoch 15 / 20) train acc: 0.458000; val_acc: 0.471000
(Iteration 3701 / 4900) loss: 1.554661
(Iteration 3801 / 4900) loss: 1.496944
(Iteration 3901 / 4900) loss: 1.535536
(Epoch 16 / 20) train acc: 0.491000; val_acc: 0.481000
(Iteration 4001 / 4900) loss: 1.479284
(Iteration 4101 / 4900) loss: 1.462916
(Epoch 17 / 20) train acc: 0.496000; val_acc: 0.496000
(Iteration 4201 / 4900) loss: 1.575096
(Iteration 4301 / 4900) loss: 1.500447
(Iteration 4401 / 4900) loss: 1.540385
(Epoch 18 / 20) train acc: 0.513000; val_acc: 0.505000
(Iteration 4501 / 4900) loss: 1.522551
(Iteration 4601 / 4900) loss: 1.557755
(Epoch 19 / 20) train acc: 0.473000; val_acc: 0.502000
(Iteration 4701 / 4900) loss: 1.472446
(Iteration 4801 / 4900) loss: 1.584254
(Epoch 20 / 20) train acc: 0.504000; val_acc: 0.506000
lr = 1.000000e-02,reg = 1.000000e-02,acc = 5.060000e-01
(Iteration 1 / 4900) loss: 2.347083
(Epoch 0 / 20) train acc: 0.099000; val_acc: 0.084000
(Iteration 101 / 4900) loss: 2.308763
(Iteration 201 / 4900) loss: 2.303468
(Epoch 1 / 20) train acc: 0.102000; val_acc: 0.078000
(Iteration 301 / 4900) loss: 2.303056
(Iteration 401 / 4900) loss: 2.302604
(Epoch 2 / 20) train acc: 0.091000; val_acc: 0.102000
(Iteration 501 / 4900) loss: 2.303017
(Iteration 601 / 4900) loss: 2.302396
(Iteration 701 / 4900) loss: 2.302518
(Epoch 3 / 20) train acc: 0.111000; val_acc: 0.087000
(Iteration 801 / 4900) loss: 2.302638
(Iteration 901 / 4900) loss: 2.302952
(Epoch 4 / 20) train acc: 0.102000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: 2.302537
(Iteration 1101 / 4900) loss: 2.302618
(Iteration 1201 / 4900) loss: 2.302439
(Epoch 5 / 20) train acc: 0.105000; val_acc: 0.087000
(Iteration 1301 / 4900) loss: 2.302294
(Iteration 1401 / 4900) loss: 2.302643
(Epoch 6 / 20) train acc: 0.122000; val_acc: 0.078000
(Iteration 1501 / 4900) loss: 2.302586
(Iteration 1601 / 4900) loss: 2.302408
```

```
(Iteration 1701 / 4900) loss: 2.302071
(Epoch 7 / 20) train acc: 0.090000; val_acc: 0.087000
(Iteration 1801 / 4900) loss: 2.303096
(Iteration 1901 / 4900) loss: 2.302676
(Epoch 8 / 20) train acc: 0.102000; val_acc: 0.079000
(Iteration 2001 / 4900) loss: 2.302010
(Iteration 2101 / 4900) loss: 2.303279
(Iteration 2201 / 4900) loss: 2.302755
(Epoch 9 / 20) train acc: 0.116000; val_acc: 0.087000
(Iteration 2301 / 4900) loss: 2.301409
(Iteration 2401 / 4900) loss: 2.302332
(Epoch 10 / 20) train acc: 0.103000; val_acc: 0.079000
(Iteration 2501 / 4900) loss: 2.302108
(Iteration 2601 / 4900) loss: 2.302335
(Epoch 11 / 20) train acc: 0.096000; val_acc: 0.078000
(Iteration 2701 / 4900) loss: 2.302237
(Iteration 2801 / 4900) loss: 2.303106
(Iteration 2901 / 4900) loss: 2.302633
(Epoch 12 / 20) train acc: 0.089000; val_acc: 0.078000
(Iteration 3001 / 4900) loss: 2.302951
(Iteration 3101 / 4900) loss: 2.302758
(Epoch 13 / 20) train acc: 0.092000; val_acc: 0.079000
(Iteration 3201 / 4900) loss: 2.302751
(Iteration 3301 / 4900) loss: 2.303498
(Iteration 3401 / 4900) loss: 2.302093
(Epoch 14 / 20) train acc: 0.086000; val_acc: 0.087000
(Iteration 3501 / 4900) loss: 2.302086
(Iteration 3601 / 4900) loss: 2.302747
(Epoch 15 / 20) train acc: 0.103000; val_acc: 0.098000
(Iteration 3701 / 4900) loss: 2.302919
(Iteration 3801 / 4900) loss: 2.303620
(Iteration 3901 / 4900) loss: 2.302546
(Epoch 16 / 20) train acc: 0.101000; val_acc: 0.098000
(Iteration 4001 / 4900) loss: 2.302851
(Iteration 4101 / 4900) loss: 2.302292
(Epoch 17 / 20) train acc: 0.112000; val_acc: 0.107000
(Iteration 4201 / 4900) loss: 2.302791
(Iteration 4301 / 4900) loss: 2.302456
(Iteration 4401 / 4900) loss: 2.302977
(Epoch 18 / 20) train acc: 0.131000; val_acc: 0.098000
(Iteration 4501 / 4900) loss: 2.303153
(Iteration 4601 / 4900) loss: 2.302388
(Epoch 19 / 20) train acc: 0.103000; val_acc: 0.098000
(Iteration 4701 / 4900) loss: 2.302266
(Iteration 4801 / 4900) loss: 2.302611
(Epoch 20 / 20) train acc: 0.097000; val_acc: 0.098000
lr = 1.000000e-02,reg = 1.000000e+00,acc = 1.070000e-01
(Iteration 1 / 4900) loss: 2.302567
```

```
(Epoch 0 / 20) train acc: 0.111000; val_acc: 0.143000
(Iteration 101 / 4900) loss: 2.301422
(Iteration 201 / 4900) loss: 2.297731
(Epoch 1 / 20) train acc: 0.228000; val_acc: 0.190000
(Iteration 301 / 4900) loss: 2.276648
(Iteration 401 / 4900) loss: 2.171996
(Epoch 2 / 20) train acc: 0.286000; val_acc: 0.290000
(Iteration 501 / 4900) loss: 2.011564
(Iteration 601 / 4900) loss: 1.909388
(Iteration 701 / 4900) loss: 1.717931
(Epoch 3 / 20) train acc: 0.398000; val_acc: 0.398000
(Iteration 801 / 4900) loss: 1.653823
(Iteration 901 / 4900) loss: 1.579910
(Epoch 4 / 20) train acc: 0.477000; val_acc: 0.438000
(Iteration 1001 / 4900) loss: 1.510102
(Iteration 1101 / 4900) loss: 1.483242
(Iteration 1201 / 4900) loss: 1.513041
(Epoch 5 / 20) train acc: 0.480000; val_acc: 0.468000
(Iteration 1301 / 4900) loss: 1.505100
(Iteration 1401 / 4900) loss: 1.365304
(Epoch 6 / 20) train acc: 0.498000; val_acc: 0.496000
(Iteration 1501 / 4900) loss: 1.448765
(Iteration 1601 / 4900) loss: 1.381322
(Iteration 1701 / 4900) loss: 1.444054
(Epoch 7 / 20) train acc: 0.510000; val_acc: 0.515000
(Iteration 1801 / 4900) loss: 1.349988
(Iteration 1901 / 4900) loss: 1.434989
(Epoch 8 / 20) train acc: 0.521000; val_acc: 0.517000
(Iteration 2001 / 4900) loss: 1.509201
(Iteration 2101 / 4900) loss: 1.264705
(Iteration 2201 / 4900) loss: 1.457419
(Epoch 9 / 20) train acc: 0.528000; val_acc: 0.524000
(Iteration 2301 / 4900) loss: 1.281672
(Iteration 2401 / 4900) loss: 1.369462
(Epoch 10 / 20) train acc: 0.529000; val_acc: 0.521000
(Iteration 2501 / 4900) loss: 1.288858
(Iteration 2601 / 4900) loss: 1.362407
(Epoch 11 / 20) train acc: 0.555000; val_acc: 0.530000
(Iteration 2701 / 4900) loss: 1.501555
(Iteration 2801 / 4900) loss: 1.312469
(Iteration 2901 / 4900) loss: 1.355646
(Epoch 12 / 20) train acc: 0.530000; val_acc: 0.529000
(Iteration 3001 / 4900) loss: 1.247890
(Iteration 3101 / 4900) loss: 1.355769
(Epoch 13 / 20) train acc: 0.529000; val_acc: 0.529000
(Iteration 3201 / 4900) loss: 1.218978
(Iteration 3301 / 4900) loss: 1.404540
(Iteration 3401 / 4900) loss: 1.238041
```

```
(Epoch 14 / 20) train acc: 0.541000; val_acc: 0.533000
(Iteration 3501 / 4900) loss: 1.329880
(Iteration 3601 / 4900) loss: 1.170476
(Epoch 15 / 20) train acc: 0.549000; val_acc: 0.544000
(Iteration 3701 / 4900) loss: 1.280084
(Iteration 3801 / 4900) loss: 1.227110
(Iteration 3901 / 4900) loss: 1.230475
(Epoch 16 / 20) train acc: 0.548000; val_acc: 0.538000
(Iteration 4001 / 4900) loss: 1.259272
(Iteration 4101 / 4900) loss: 1.143284
(Epoch 17 / 20) train acc: 0.540000; val_acc: 0.548000
(Iteration 4201 / 4900) loss: 1.232617
(Iteration 4301 / 4900) loss: 1.219501
(Iteration 4401 / 4900) loss: 1.271327
(Epoch 18 / 20) train acc: 0.542000; val_acc: 0.549000
(Iteration 4501 / 4900) loss: 1.293902
(Iteration 4601 / 4900) loss: 1.373365
(Epoch 19 / 20) train acc: 0.546000; val_acc: 0.548000
(Iteration 4701 / 4900) loss: 1.270895
(Iteration 4801 / 4900) loss: 1.230224
(Epoch 20 / 20) train acc: 0.551000; val_acc: 0.555000
lr = 2.750000e-02,reg = 1.000000e-06,acc = 5.550000e-01
(Iteration 1 / 4900) loss: 2.303065
(Epoch 0 / 20) train acc: 0.097000; val_acc: 0.108000
(Iteration 101 / 4900) loss: 2.301857
(Iteration 201 / 4900) loss: 2.298892
(Epoch 1 / 20) train acc: 0.228000; val_acc: 0.237000
(Iteration 301 / 4900) loss: 2.283837
(Iteration 401 / 4900) loss: 2.220051
(Epoch 2 / 20) train acc: 0.302000; val_acc: 0.303000
(Iteration 501 / 4900) loss: 2.040105
(Iteration 601 / 4900) loss: 1.942146
(Iteration 701 / 4900) loss: 1.736899
(Epoch 3 / 20) train acc: 0.368000; val_acc: 0.370000
(Iteration 801 / 4900) loss: 1.742202
(Iteration 901 / 4900) loss: 1.657665
(Epoch 4 / 20) train acc: 0.426000; val_acc: 0.418000
(Iteration 1001 / 4900) loss: 1.631184
(Iteration 1101 / 4900) loss: 1.553225
(Iteration 1201 / 4900) loss: 1.566969
(Epoch 5 / 20) train acc: 0.508000; val_acc: 0.458000
(Iteration 1301 / 4900) loss: 1.588420
(Iteration 1401 / 4900) loss: 1.465972
(Epoch 6 / 20) train acc: 0.490000; val_acc: 0.490000
(Iteration 1501 / 4900) loss: 1.492067
(Iteration 1601 / 4900) loss: 1.454353
(Iteration 1701 / 4900) loss: 1.533021
(Epoch 7 / 20) train acc: 0.512000; val_acc: 0.494000
```

```
(Iteration 1801 / 4900) loss: 1.378880
(Iteration 1901 / 4900) loss: 1.527636
(Epoch 8 / 20) train acc: 0.502000; val_acc: 0.515000
(Iteration 2001 / 4900) loss: 1.440610
(Iteration 2101 / 4900) loss: 1.486267
(Iteration 2201 / 4900) loss: 1.479298
(Epoch 9 / 20) train acc: 0.522000; val_acc: 0.519000
(Iteration 2301 / 4900) loss: 1.434737
(Iteration 2401 / 4900) loss: 1.542681
(Epoch 10 / 20) train acc: 0.531000; val_acc: 0.520000
(Iteration 2501 / 4900) loss: 1.345520
(Iteration 2601 / 4900) loss: 1.419700
(Epoch 11 / 20) train acc: 0.532000; val_acc: 0.527000
(Iteration 2701 / 4900) loss: 1.428515
(Iteration 2801 / 4900) loss: 1.426800
(Iteration 2901 / 4900) loss: 1.359122
(Epoch 12 / 20) train acc: 0.517000; val_acc: 0.530000
(Iteration 3001 / 4900) loss: 1.367989
(Iteration 3101 / 4900) loss: 1.478787
(Epoch 13 / 20) train acc: 0.533000; val_acc: 0.525000
(Iteration 3201 / 4900) loss: 1.479545
(Iteration 3301 / 4900) loss: 1.415893
(Iteration 3401 / 4900) loss: 1.362511
(Epoch 14 / 20) train acc: 0.542000; val_acc: 0.518000
(Iteration 3501 / 4900) loss: 1.399797
(Iteration 3601 / 4900) loss: 1.505351
(Epoch 15 / 20) train acc: 0.517000; val_acc: 0.523000
(Iteration 3701 / 4900) loss: 1.532121
(Iteration 3801 / 4900) loss: 1.420772
(Iteration 3901 / 4900) loss: 1.430865
(Epoch 16 / 20) train acc: 0.554000; val_acc: 0.525000
(Iteration 4001 / 4900) loss: 1.549768
(Iteration 4101 / 4900) loss: 1.446778
(Epoch 17 / 20) train acc: 0.537000; val_acc: 0.533000
(Iteration 4201 / 4900) loss: 1.415636
(Iteration 4301 / 4900) loss: 1.449086
(Iteration 4401 / 4900) loss: 1.513283
(Epoch 18 / 20) train acc: 0.511000; val_acc: 0.526000
(Iteration 4501 / 4900) loss: 1.358274
(Iteration 4601 / 4900) loss: 1.579990
(Epoch 19 / 20) train acc: 0.535000; val_acc: 0.518000
(Iteration 4701 / 4900) loss: 1.477078
(Iteration 4801 / 4900) loss: 1.417142
(Epoch 20 / 20) train acc: 0.542000; val_acc: 0.529000
lr = 2.750000e-02,reg = 1.000000e-02,acc = 5.330000e-01
(Iteration 1 / 4900) loss: 2.347287
(Epoch 0 / 20) train acc: 0.084000; val_acc: 0.113000
(Iteration 101 / 4900) loss: 2.302581
```

```
(Iteration 201 / 4900) loss: 2.302160
(Epoch 1 / 20) train acc: 0.085000; val_acc: 0.078000
(Iteration 301 / 4900) loss: 2.302775
(Iteration 401 / 4900) loss: 2.303063
(Epoch 2 / 20) train acc: 0.085000; val_acc: 0.107000
(Iteration 501 / 4900) loss: 2.302698
(Iteration 601 / 4900) loss: 2.303253
(Iteration 701 / 4900) loss: 2.302841
(Epoch 3 / 20) train acc: 0.119000; val_acc: 0.119000
(Iteration 801 / 4900) loss: 2.303719
(Iteration 901 / 4900) loss: 2.303465
(Epoch 4 / 20) train acc: 0.088000; val_acc: 0.079000
(Iteration 1001 / 4900) loss: 2.302526
(Iteration 1101 / 4900) loss: 2.302159
(Iteration 1201 / 4900) loss: 2.303090
(Epoch 5 / 20) train acc: 0.096000; val_acc: 0.079000
(Iteration 1301 / 4900) loss: 2.302959
(Iteration 1401 / 4900) loss: 2.303402
(Epoch 6 / 20) train acc: 0.094000; val_acc: 0.079000
(Iteration 1501 / 4900) loss: 2.302606
(Iteration 1601 / 4900) loss: 2.302573
(Iteration 1701 / 4900) loss: 2.301749
(Epoch 7 / 20) train acc: 0.110000; val_acc: 0.078000
(Iteration 1801 / 4900) loss: 2.303293
(Iteration 1901 / 4900) loss: 2.302696
(Epoch 8 / 20) train acc: 0.113000; val_acc: 0.107000
(Iteration 2001 / 4900) loss: 2.303485
(Iteration 2101 / 4900) loss: 2.302699
(Iteration 2201 / 4900) loss: 2.303614
(Epoch 9 / 20) train acc: 0.084000; val_acc: 0.107000
(Iteration 2301 / 4900) loss: 2.302488
(Iteration 2401 / 4900) loss: 2.302829
(Epoch 10 / 20) train acc: 0.095000; val_acc: 0.107000
(Iteration 2501 / 4900) loss: 2.303396
(Iteration 2601 / 4900) loss: 2.302178
(Epoch 11 / 20) train acc: 0.078000; val_acc: 0.078000
(Iteration 2701 / 4900) loss: 2.303332
(Iteration 2801 / 4900) loss: 2.302887
(Iteration 2901 / 4900) loss: 2.302503
(Epoch 12 / 20) train acc: 0.109000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: 2.301942
(Iteration 3101 / 4900) loss: 2.302528
(Epoch 13 / 20) train acc: 0.089000; val_acc: 0.102000
(Iteration 3201 / 4900) loss: 2.303193
(Iteration 3301 / 4900) loss: 2.301871
(Iteration 3401 / 4900) loss: 2.302977
(Epoch 14 / 20) train acc: 0.103000; val_acc: 0.087000
(Iteration 3501 / 4900) loss: 2.302995
```

```
(Iteration 3601 / 4900) loss: 2.301195
(Epoch 15 / 20) train acc: 0.112000; val_acc: 0.087000
(Iteration 3701 / 4900) loss: 2.302434
(Iteration 3801 / 4900) loss: 2.302355
(Iteration 3901 / 4900) loss: 2.303027
(Epoch 16 / 20) train acc: 0.097000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: 2.301919
(Iteration 4101 / 4900) loss: 2.302523
(Epoch 17 / 20) train acc: 0.113000; val_acc: 0.078000
(Iteration 4201 / 4900) loss: 2.302703
(Iteration 4301 / 4900) loss: 2.303665
(Iteration 4401 / 4900) loss: 2.303587
(Epoch 18 / 20) train acc: 0.100000; val_acc: 0.112000
(Iteration 4501 / 4900) loss: 2.302452
(Iteration 4601 / 4900) loss: 2.302173
(Epoch 19 / 20) train acc: 0.085000; val_acc: 0.098000
(Iteration 4701 / 4900) loss: 2.302429
(Iteration 4801 / 4900) loss: 2.302753
(Epoch 20 / 20) train acc: 0.103000; val_acc: 0.078000
lr = 2.750000e-02,reg = 1.000000e+00,acc = 1.190000e-01
```

[30]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 58% accuracy. It is also possible to get >60% accuracy
# with careful tuning.
y_test_pred = np.argmax(best_net.model.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.536
```

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

[ ]:
```

```

FullyConnectedNets

July 16, 2025

1 Multi-Layer Fully Connected Network

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

[21]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

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

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

[22]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

%load_ext autoreload
%autoreload 2

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

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

```
%reload_ext autoreload
```

[23]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{k}: {v.shape}")
```

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

1.1 Initial Loss and Gradient Check

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

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

[24]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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

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

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

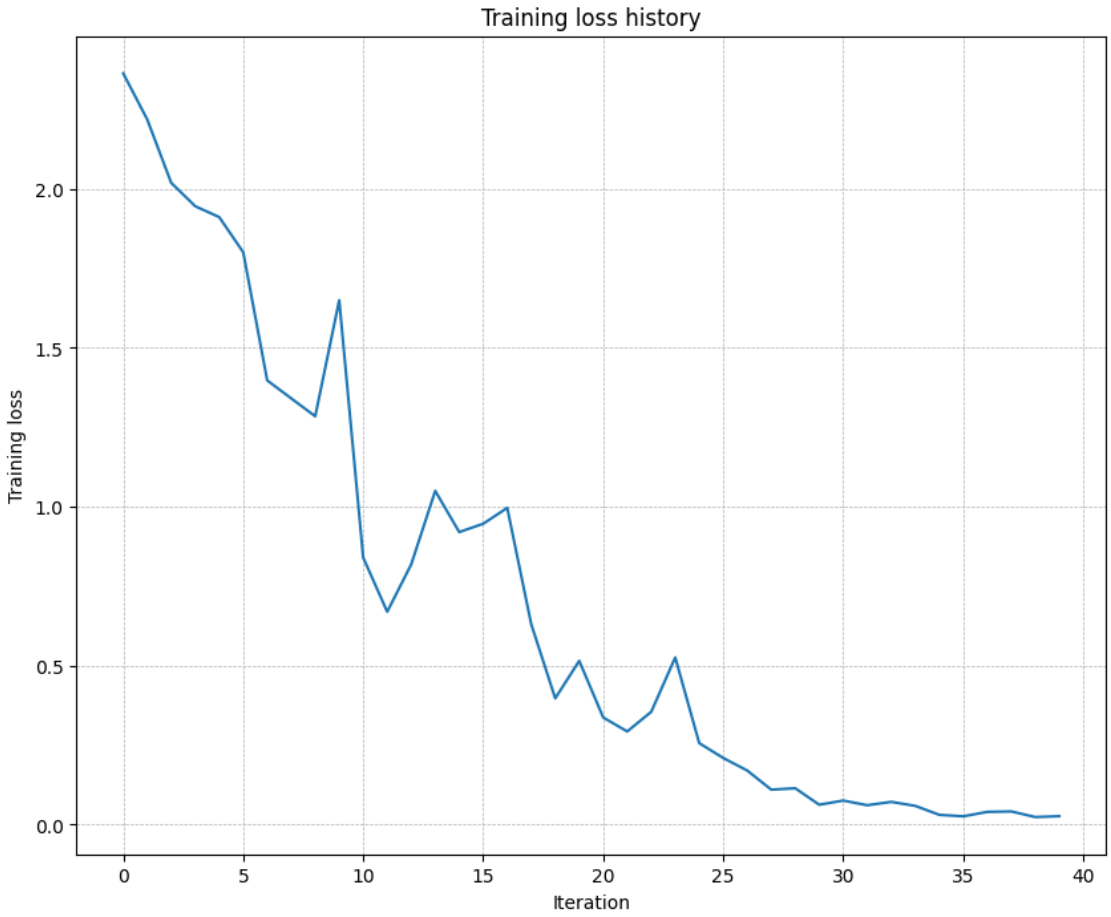
```
Running check with reg =  0
Initial loss:  2.300479089768492
W1 relative error: 1.0252674471656573e-07
W2 relative error: 2.2120479295080622e-05
W3 relative error: 4.5623278736665505e-07
b1 relative error: 4.6600944653202505e-09
b2 relative error: 2.085654276112763e-09
b3 relative error: 1.689724888469736e-10
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 1.1358395917166688e-08
W2 relative error: 6.86942277940646e-08
W3 relative error: 3.483989247437803e-08
b1 relative error: 1.475242847895799e-08
b2 relative error: 1.7223751746766738e-09
b3 relative error: 2.378772438198909e-10
```

As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy within 20 epochs.

[25]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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

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


model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
```

```
    update_rule="sgd",
    optim_config={"learning_rate": learning_rate},
)
solver.train()

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

```
(Iteration 1 / 40) loss: 2.363364
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.108000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.127000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.172000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.184000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.181000
(Epoch 5 / 20) train acc: 0.740000; val_acc: 0.190000
(Iteration 11 / 40) loss: 0.839976
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.187000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.183000
(Epoch 8 / 20) train acc: 0.820000; val_acc: 0.177000
(Epoch 9 / 20) train acc: 0.860000; val_acc: 0.200000
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.191000
(Iteration 21 / 40) loss: 0.337174
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.189000
(Epoch 12 / 20) train acc: 0.940000; val_acc: 0.180000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.195000
(Iteration 31 / 40) loss: 0.075911
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.182000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.201000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.207000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.185000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.192000
```



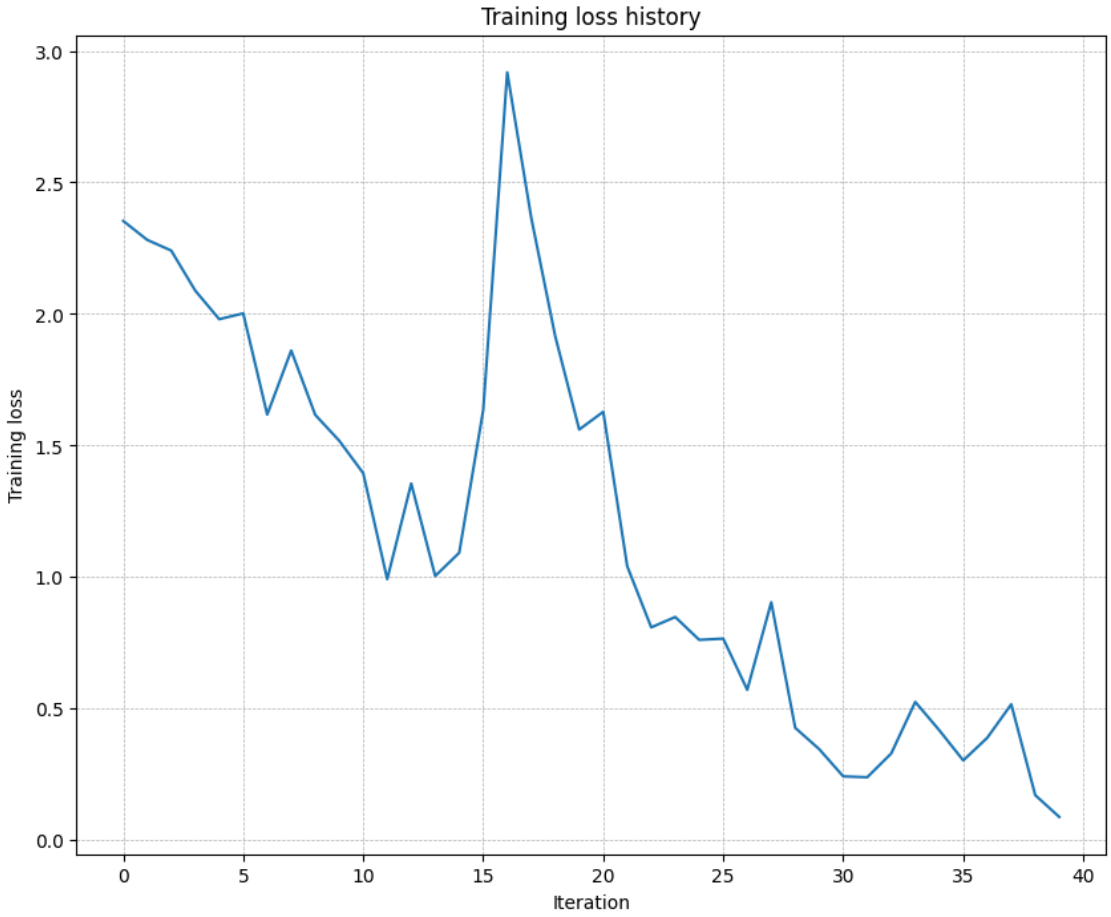
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[26]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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

learning_rate = 0.04 # Experiment with this!
weight_scale = 0.03  # Experiment with this!
```

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

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

```
(Iteration 1 / 40) loss: 2.353765
(Epoch 0 / 20) train acc: 0.380000; val_acc: 0.124000
(Epoch 1 / 20) train acc: 0.180000; val_acc: 0.080000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.119000
(Epoch 3 / 20) train acc: 0.300000; val_acc: 0.133000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.173000
(Epoch 5 / 20) train acc: 0.420000; val_acc: 0.124000
(Iteration 11 / 40) loss: 1.395147
(Epoch 6 / 20) train acc: 0.700000; val_acc: 0.175000
(Epoch 7 / 20) train acc: 0.580000; val_acc: 0.133000
(Epoch 8 / 20) train acc: 0.260000; val_acc: 0.082000
(Epoch 9 / 20) train acc: 0.280000; val_acc: 0.128000
(Epoch 10 / 20) train acc: 0.480000; val_acc: 0.134000
(Iteration 21 / 40) loss: 1.628341
(Epoch 11 / 20) train acc: 0.820000; val_acc: 0.169000
(Epoch 12 / 20) train acc: 0.800000; val_acc: 0.174000
(Epoch 13 / 20) train acc: 0.700000; val_acc: 0.118000
(Epoch 14 / 20) train acc: 0.880000; val_acc: 0.166000
(Epoch 15 / 20) train acc: 0.880000; val_acc: 0.184000
(Iteration 31 / 40) loss: 0.242815
(Epoch 16 / 20) train acc: 0.920000; val_acc: 0.194000
(Epoch 17 / 20) train acc: 0.920000; val_acc: 0.169000
(Epoch 18 / 20) train acc: 0.880000; val_acc: 0.174000
(Epoch 19 / 20) train acc: 0.920000; val_acc: 0.148000
```

```
(Epoch 20 / 20) train acc: 0.960000; val_acc: 0.164000
```



1.2 Inline Question 1:

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

1.3 Answer:

5-layers is more sensitive.

2 Update rules

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

2.1 SGD+Momentum

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

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

[27]:
```
from cs231n.optim import sgd_momentum

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

config = {"learning_rate": 1e-3, "velocity": v}
next_w, _ = sgd_momentum(w, dw, config=config)
# print(next_w)
expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

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

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

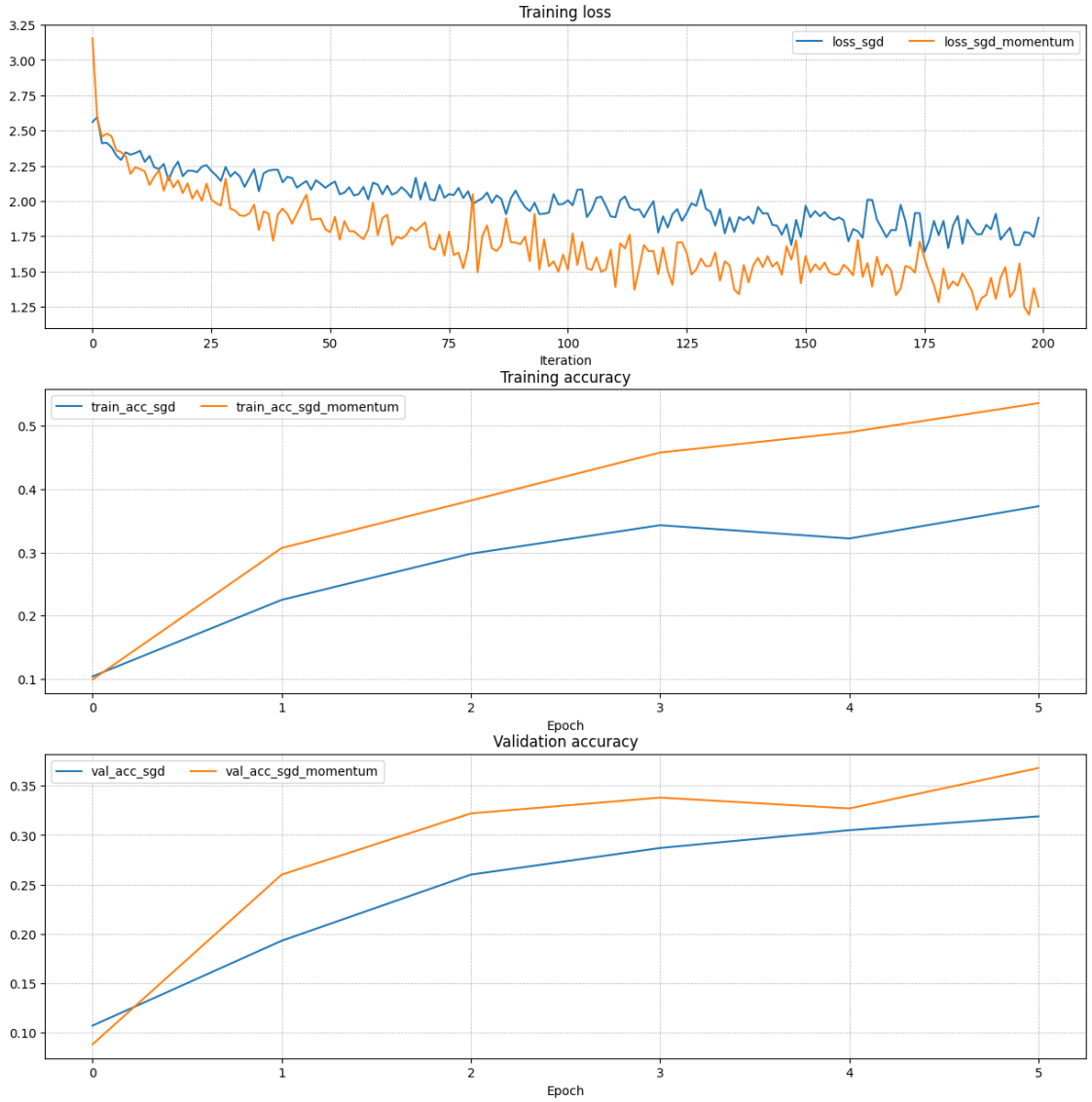
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[28]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}
```

```
solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
    print('Running with ', update_rule)
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=5e-2
    )
    # print("created fcn")
    solver = Solver(
        model,
        small_data,
        num_epochs=5,
        batch_size=100,
        update_rule=update_rule,
        optim_config={'learning_rate': 5e-3},
        verbose=True,
    )
    # print("created solver")
    solvers[update_rule] = solver
    solver.train()

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

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

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

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

plt.show()
```

```
Running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.107000
(Iteration 11 / 200) loss: 2.356070
(Iteration 21 / 200) loss: 2.214091
(Iteration 31 / 200) loss: 2.205928
```

```
(Epoch 1 / 5) train acc: 0.225000; val_acc: 0.193000
(Iteration 41 / 200) loss: 2.132095
(Iteration 51 / 200) loss: 2.118950
(Iteration 61 / 200) loss: 2.116443
(Iteration 71 / 200) loss: 2.132549
(Epoch 2 / 5) train acc: 0.298000; val_acc: 0.260000
(Iteration 81 / 200) loss: 1.977227
(Iteration 91 / 200) loss: 2.007528
(Iteration 101 / 200) loss: 2.004762
(Iteration 111 / 200) loss: 1.885342
(Epoch 3 / 5) train acc: 0.343000; val_acc: 0.287000
(Iteration 121 / 200) loss: 1.891517
(Iteration 131 / 200) loss: 1.923677
(Iteration 141 / 200) loss: 1.957743
(Iteration 151 / 200) loss: 1.966736
(Epoch 4 / 5) train acc: 0.322000; val_acc: 0.305000
(Iteration 161 / 200) loss: 1.801483
(Iteration 171 / 200) loss: 1.973780
(Iteration 181 / 200) loss: 1.666572
(Iteration 191 / 200) loss: 1.909494
(Epoch 5 / 5) train acc: 0.373000; val_acc: 0.319000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153778
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.088000
(Iteration 11 / 200) loss: 2.227203
(Iteration 21 / 200) loss: 2.125706
(Iteration 31 / 200) loss: 1.932695
(Epoch 1 / 5) train acc: 0.307000; val_acc: 0.260000
(Iteration 41 / 200) loss: 1.946488
(Iteration 51 / 200) loss: 1.778584
(Iteration 61 / 200) loss: 1.758119
(Iteration 71 / 200) loss: 1.849137
(Epoch 2 / 5) train acc: 0.382000; val_acc: 0.322000
(Iteration 81 / 200) loss: 2.048671
(Iteration 91 / 200) loss: 1.693223
(Iteration 101 / 200) loss: 1.511693
(Iteration 111 / 200) loss: 1.390754
(Epoch 3 / 5) train acc: 0.458000; val_acc: 0.338000
(Iteration 121 / 200) loss: 1.670614
(Iteration 131 / 200) loss: 1.540271
(Iteration 141 / 200) loss: 1.597365
(Iteration 151 / 200) loss: 1.609851
(Epoch 4 / 5) train acc: 0.490000; val_acc: 0.327000
(Iteration 161 / 200) loss: 1.472687
(Iteration 171 / 200) loss: 1.378620
(Iteration 181 / 200) loss: 1.378175
(Iteration 191 / 200) loss: 1.305935
(Epoch 5 / 5) train acc: 0.536000; val_acc: 0.368000
```



2.2 RMSProp and Adam

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

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

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

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

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

[29]:
```
# Test RMSProp implementation
from cs231n.optim import rmsprop

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

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

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[29], line 24
     17 expected_cache = np.asarray([
     18   [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
     19   [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
     20   [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
     21   [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])
     23 # You should see relative errors around e-7 or less
---> 24 print('next_w error: ', rel_error(expected_next_w, next_w))
     25 print('cache error: ', rel_error(expected_cache, config['cache']))

Cell In[22], line 20, in rel_error(x, y)
     18 def rel_error(x, y):
     19     """Returns relative error."""
---> 20     return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.
 ↪abs(y))))
```

```
TypeError: unsupported operand type(s) for -: 'float' and 'NoneType'
```

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

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

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

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

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

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

[ ]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
    print('Running with ', update_rule)
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=5e-2
    )
    solver = Solver(
        model,
```

```
        small_data,
        num_epochs=5,
        batch_size=100,
        update_rule=update_rule,
        optim_config={'learning_rate': learning_rates[update_rule]},
        verbose=True
    )
    solvers[update_rule] = solver
    solver.train()
    print()

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

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

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

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

plt.show()
```

2.3 Inline Question 2:

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

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

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

2.4 Answer:

[FILL THIS IN]

3 Train a Good Model!

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

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

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

[ ]:
```
best_model = None

################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****



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

4 Test Your Model!

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

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