knn

September 12, 2020

[1]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'cs231n/assignment1/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-09-09 02:36:10--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  44.4MB/s    in 5.4s

2020-09-09 02:36:16 (30.1 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

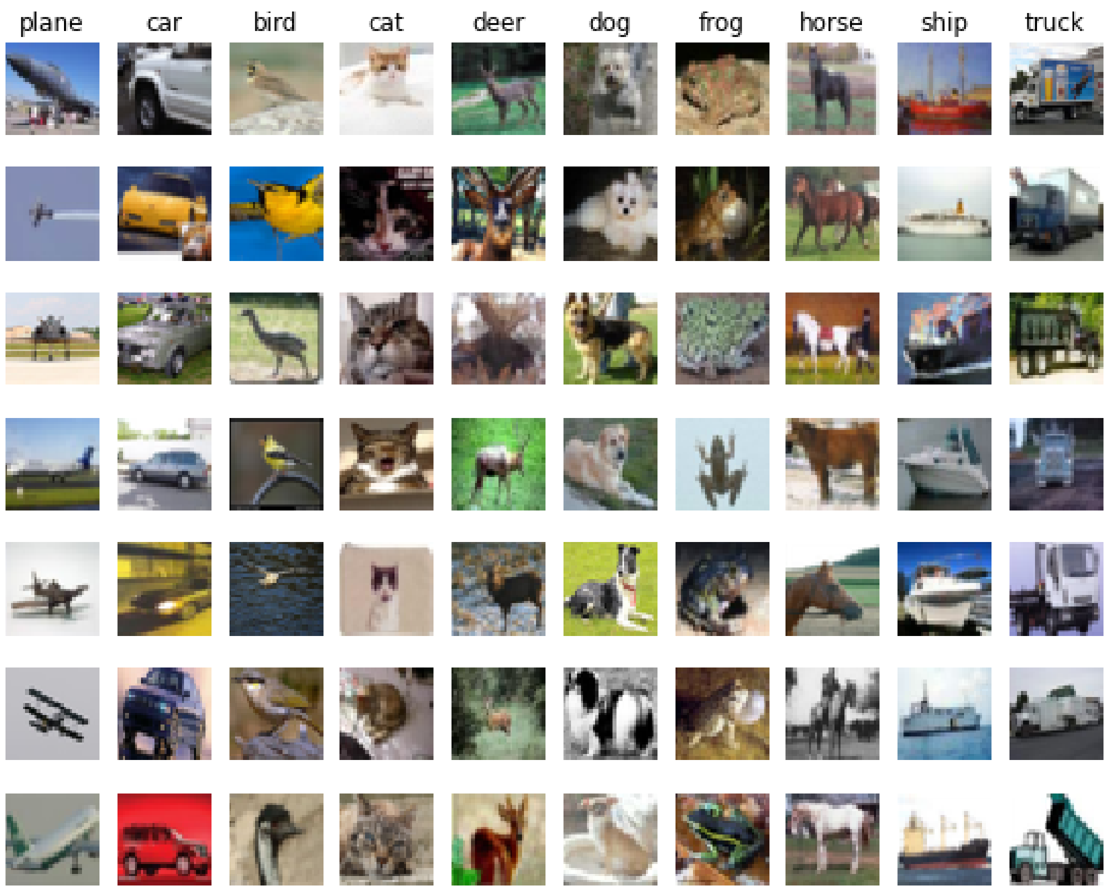
[2]:
```
# Run some setup code for this notebook.

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

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

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

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

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

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

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

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



[5]:
```python
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

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

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

[6]:
```
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.

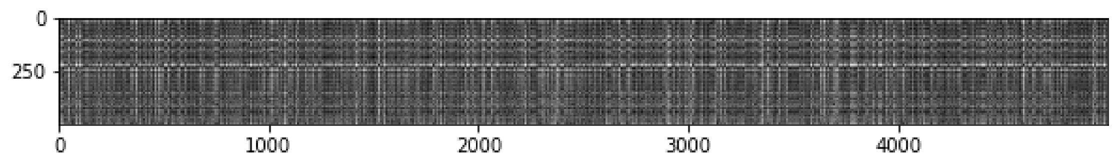
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

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

[8]:
```
# 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 visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

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

YourAnswer : Distinctly bright rows occur if there is a high distance for the row. This means that the ith training image is very different from the testing images. Distinctly bright columns occur if a training image is distant from all test images.

[9]:
```python
# 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`:

[10]:
```python
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

$\color{blue}{Your Answer:}$ 2, 4, 5

$\color{blue}{Your Explanation:}$ 2 and 4 have to do with pixel-wise differences, which is what the L1 distance is based on. 5 is rotation, which doesn't affect distances if everything is rotated. 1 and 3 use the overall mean which is different than the pixel-wise metric that is the L1 distance.

[11]:
```
# 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
```

[12]:
```
# 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
```

[13]:
```
# 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 31.740621 seconds
One loop version took 29.752187 seconds
No loop version took 0.468552 seconds
```

1.0.1 Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

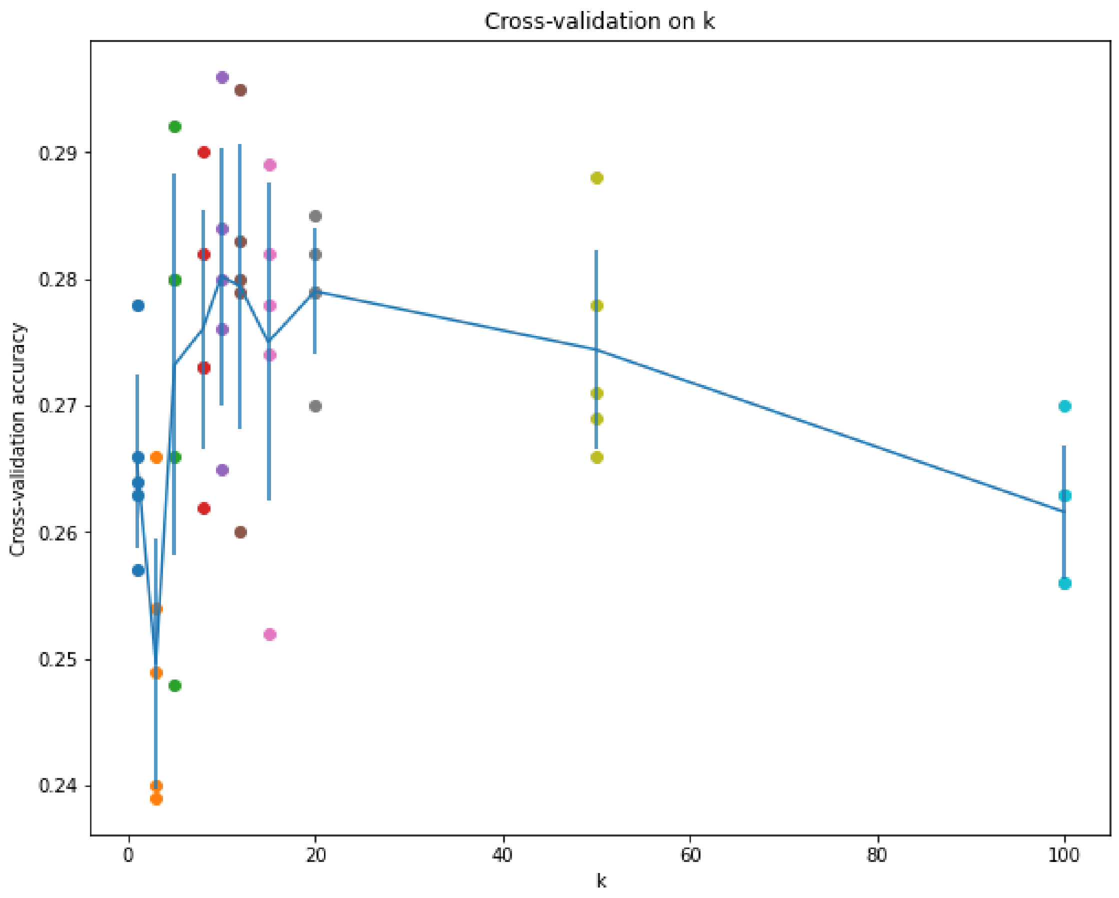
[14]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        ␣
↪#
# Split up the training data into folds. After splitting, X_train_folds and    ␣
↪#
# y_train_folds should each be lists of length num_folds, where                ␣
↪#
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     ␣
↪#
```

```
# Hint: Look up the numpy array_split function.                              ␣
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

X_train_folds = np.split(X_train, num_folds)
y_train_folds = np.split(y_train, num_folds)

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

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


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

for k in k_choices:
  acc = []
  for i in range(num_folds):
    xtr = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
    ytr = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])
    xte = X_train_folds[i]
    yte = y_train_folds[i]
    classifier.train(xtr,ytr)
    d = classifier.compute_distances_no_loops(xte)
    test_pred = classifier.predict_labels(d, k=k)
    accuracy = np.sum(test_pred == yte)
    accuracy /= float(yte.shape[0])
    acc.append(accuracy)
  k_to_accuracies.update({k:acc})


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

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

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
```

```
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

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



[16]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

$Your Answer$: 4

$Your Explanation$: It takes longer to compute all the differences to classify the test example if there is a large training set because it takes distances to all the training images.

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[17]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/k_nearest_neighbor.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```

svm

September 12, 2020

[2]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'cs231n/assignment1/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&scope=email%20https%3a%2f%2fwww.googl
eapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%2
0https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2
fwww.googleapis.com%2fauth%2fpeopleapi.readonly&response_type=code

Enter your authorization code:
4/4AERSFThxIRNdRv-n9RISzw7TKhoVXtXMx1bBqPfg6DbmEsmdYRWn7o
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-09-10 01:59:56--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
```

```
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  24.5MB/s    in 8.0s

2020-09-10 02:00:04 (20.2 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

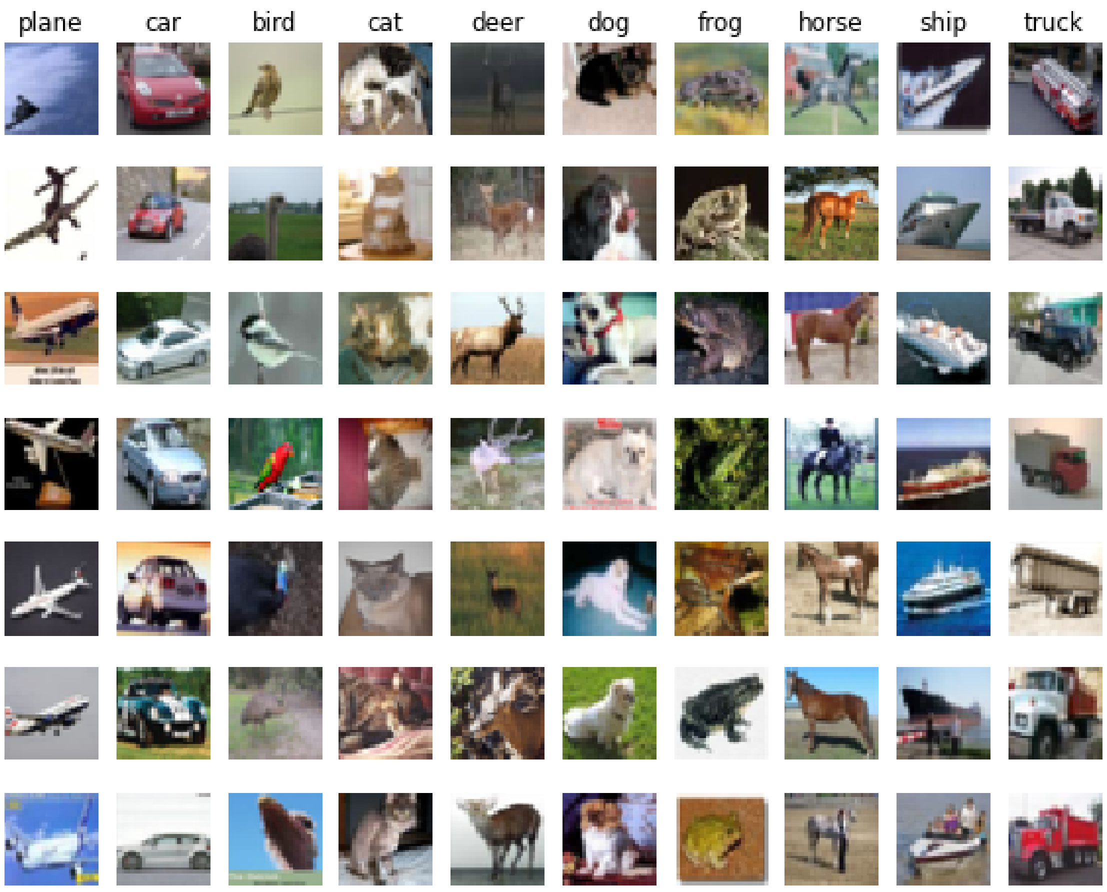
[3]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

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



[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)
```

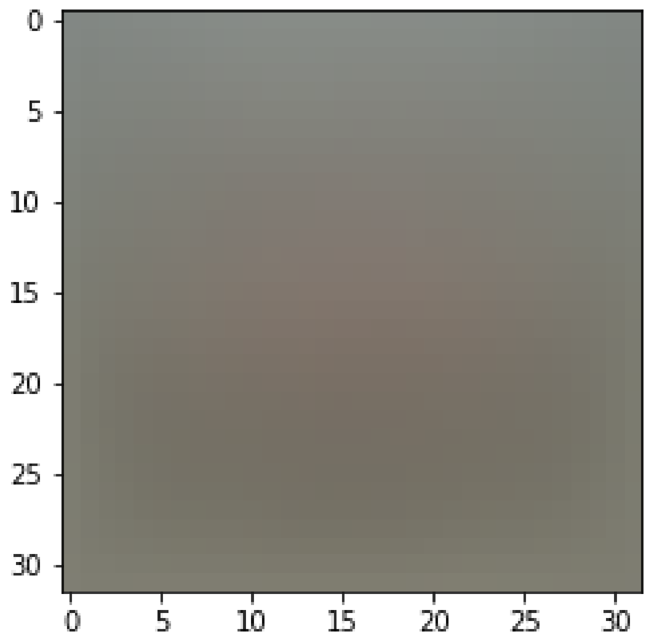
```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[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 SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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

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



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

1.2 SVM Classifier

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

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

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

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

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

```
loss: 9.051041
```

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

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

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

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

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

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

```
numerical: 19.963664 analytic: 19.963664, relative error: 1.327450e-11
numerical: 21.843689 analytic: 21.843689, relative error: 3.678162e-12
```

```
numerical: -28.809653 analytic: -28.809653, relative error: 8.638156e-12
numerical: 2.727368 analytic: 2.727368, relative error: 1.912285e-10
numerical: -6.235606 analytic: -6.330575, relative error: 7.557473e-03
numerical: 3.383283 analytic: 3.383283, relative error: 7.069610e-11
numerical: -50.060280 analytic: -50.146787, relative error: 8.632820e-04
numerical: 14.440070 analytic: 14.440070, relative error: 3.233101e-12
numerical: 28.897318 analytic: 28.897318, relative error: 8.240417e-12
numerical: 16.466067 analytic: 16.466067, relative error: 1.638134e-11
numerical: 14.468394 analytic: 14.468394, relative error: 1.281869e-11
numerical: -11.982022 analytic: -11.982022, relative error: 1.604204e-11
numerical: 14.991262 analytic: 15.073662, relative error: 2.740723e-03
numerical: 22.060758 analytic: 22.060758, relative error: 3.137183e-12
numerical: 3.752190 analytic: 3.752190, relative error: 4.288493e-11
numerical: 8.560843 analytic: 8.560843, relative error: 4.773665e-11
numerical: 34.393273 analytic: 34.480900, relative error: 1.272278e-03
numerical: -21.266496 analytic: -21.266496, relative error: 8.511546e-13
numerical: 15.097637 analytic: 15.097637, relative error: 2.317261e-11
numerical: -9.760501 analytic: -9.760501, relative error: 2.683672e-11
```

Inline Question 1

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

$\color{blue}{\textit{Your Answer:}}$ Hinge loss becomes non differentaible when any score is exactly equal to the margin more than the correct score and this causes the discrepancy.

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

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

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

```
Naive loss: 9.051041e+00 computed in 0.068717s
Vectorized loss: 9.051041e+00 computed in 0.008344s
difference: -0.000000
```

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

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

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

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

```
Naive loss and gradient: computed in 0.065870s
Vectorized loss and gradient: computed in 0.007352s
difference: 295.405637
```

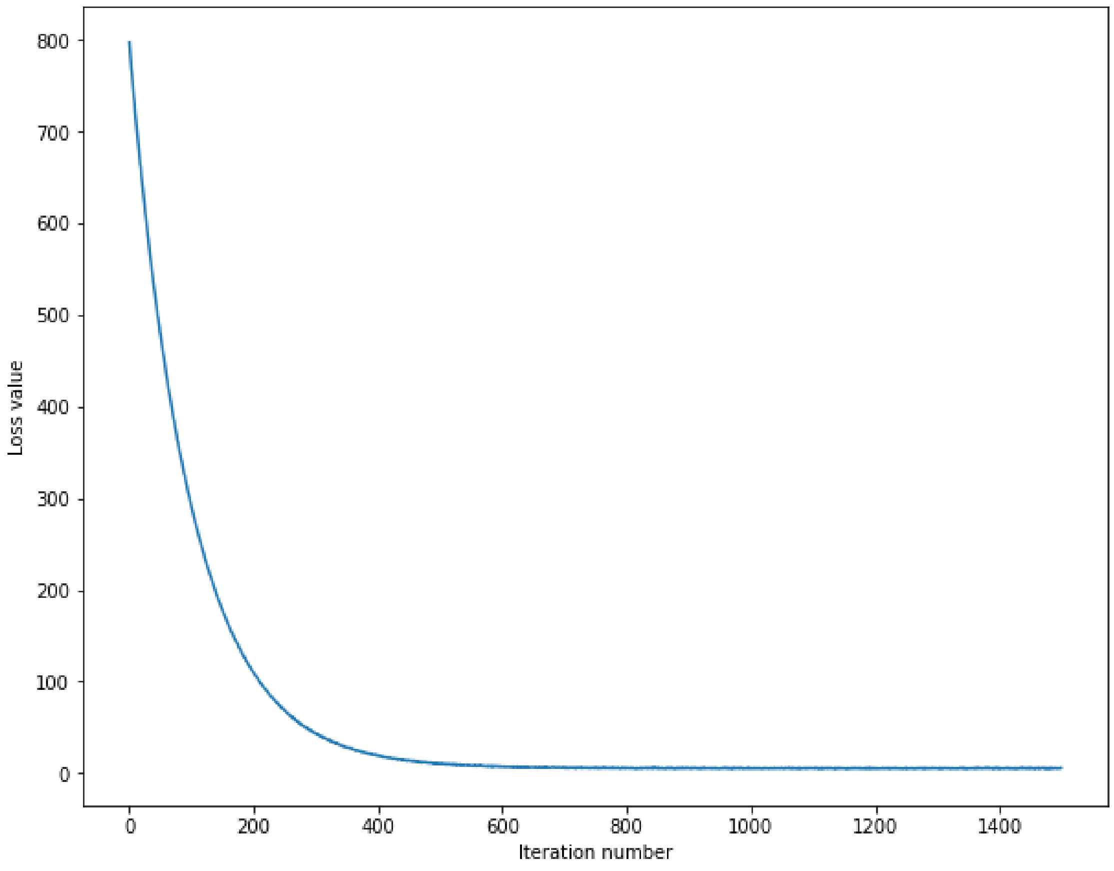
1.2.1 Stochastic Gradient Descent

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 LinearSVM
      svm = LinearSVM()
      tic = time.time()
      loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                            num_iters=1500, verbose=True)
      toc = time.time()
      print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 797.361680
iteration 100 / 1500: loss 289.242687
iteration 200 / 1500: loss 108.665423
iteration 300 / 1500: loss 43.403584
iteration 400 / 1500: loss 19.238603
iteration 500 / 1500: loss 10.808759
iteration 600 / 1500: loss 7.207775
iteration 700 / 1500: loss 6.213368
iteration 800 / 1500: loss 5.764141
iteration 900 / 1500: loss 5.435578
iteration 1000 / 1500: loss 5.099025
iteration 1100 / 1500: loss 5.410729
iteration 1200 / 1500: loss 5.180644
iteration 1300 / 1500: loss 5.082375
iteration 1400 / 1500: loss 5.789636
That took 5.107443s
```

[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 LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
```

```
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.365061
validation accuracy: 0.374000
```

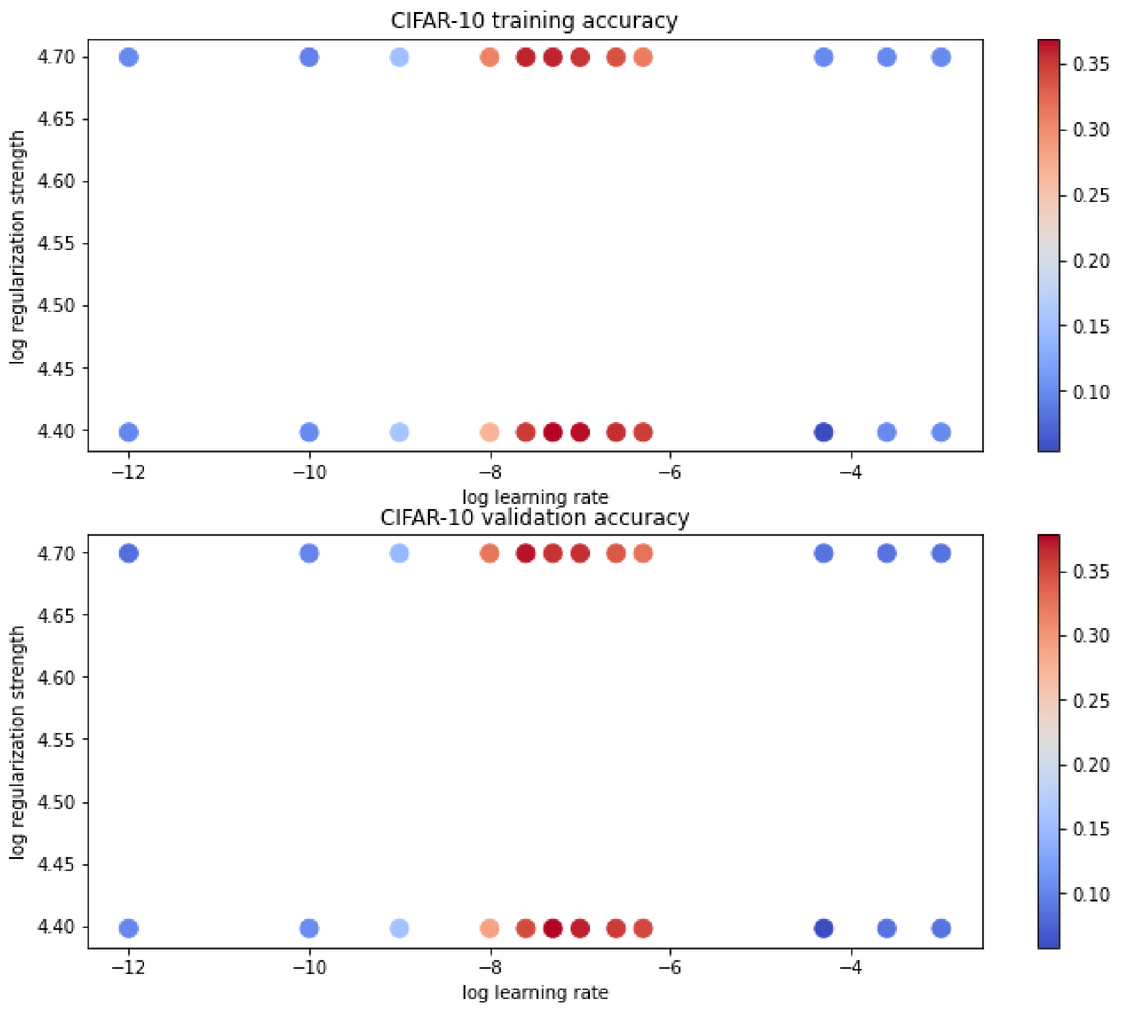
[15]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

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

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

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

```
# confident that your validation code works, you should rerun the validation ␣
↪#
# code with a larger value for num_iters.                                      ␣
↪#
################################################################################

# Provided as a reference. You may or may not want to change these␣
↪hyperparameters
learning_rates = [1e-7, 5e-5, 1e-8, 1e-3, 5e-8, 5e-7, 2.5e-4, 2.5e-8, 2.5e-7,␣
↪1e-10, 1e-12, 1e-9]
regularization_strengths = [2.5e4, 5e4]

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

for lr in learning_rates:
  for rs in regularization_strengths:
    svm = LinearSVM()
    loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,␣
↪num_iters=1500, verbose=True)
    #loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,␣
↪num_iters=1500)
    train_pred = svm.predict(X_train)
    val_pred = svm.predict(X_val)
    train_acc = np.mean(train_pred == y_train)
    val_acc = np.mean(val_pred == y_val)
    results[(lr,rs)] = (train_acc, val_acc)
    if val_acc > best_val:
      best_val = val_acc
      best_svm = svm

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

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

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

```
iteration 0 / 1500: loss 797.465222
iteration 100 / 1500: loss 289.567497
iteration 200 / 1500: loss 109.160191
iteration 300 / 1500: loss 43.266853
iteration 400 / 1500: loss 18.954836
iteration 500 / 1500: loss 10.538708
iteration 600 / 1500: loss 7.735779
iteration 700 / 1500: loss 5.743218
iteration 800 / 1500: loss 5.717361
iteration 900 / 1500: loss 5.584876
iteration 1000 / 1500: loss 6.110636
iteration 1100 / 1500: loss 5.770490
iteration 1200 / 1500: loss 5.302385
iteration 1300 / 1500: loss 5.061099
iteration 1400 / 1500: loss 5.490917
iteration 0 / 1500: loss 1562.192726
iteration 100 / 1500: loss 211.077643
iteration 200 / 1500: loss 32.434644
iteration 300 / 1500: loss 8.996359
iteration 400 / 1500: loss 5.888243
iteration 500 / 1500: loss 5.739112
iteration 600 / 1500: loss 6.010891
iteration 700 / 1500: loss 5.725783
iteration 800 / 1500: loss 5.667617
iteration 900 / 1500: loss 5.411741
iteration 1000 / 1500: loss 5.364853
iteration 1100 / 1500: loss 5.387870
iteration 1200 / 1500: loss 5.546234
iteration 1300 / 1500: loss 5.839516
iteration 1400 / 1500: loss 6.000070
iteration 0 / 1500: loss 790.977042
iteration 100 / 1500: loss 443288274973605768571361455591391232000.000000
iteration 200 / 1500: loss 7327200159228428631864304036037139779729568337321O311
338768789459483754496.000000
iteration 300 / 1500: loss 12111275033519391994439321899425918199358032502911706
081201090111128326464223607195477085750038920148482523136.000000
iteration 400 / 1500: loss 20018967647936646014779861118738801828327727325318570
53086804334680202075320999796020033585301739846444951265691048050310555461965329
297387290624.000000
iteration 500 / 1500: loss 33089750218699991195564981416727062124136553158886992
86921180467868469404927004166072900162587768023784902101619175662714539543070402
56786394299690206878559335343527131655545290752.000000
iteration 600 / 1500: loss 54694706979498555395763745552850008059272748017674931
34875631654228785750171714590190159529405211335406829849575347670796414750711574
59297252738785967423796051818128842359381582479475465101671016401787898333432381
44.000000
iteration 700 / 1500: loss 90405970181141380549637827013387031113641524290703662
45515396483879285870959455179209785882461497864702386957920918247965441266903127
26346191745933806304339636015851094057330636435370755549582356095941481130230647
8724229683160336103623655418950057984.000000
iteration 800 / 1500: loss 14943382816653601977958234255312361003857780253004384
76258633319853277534371115581673526931125607331174113912147328944800852326492275
25755049261750578740430500706530121216954696900127464744717346612989240902656043
1049393899809215629736225314824901532503954499696883156873578380768837632.000000
```

```
/content/cs231n/classifiers/linear_svm.py:92: RuntimeWarning: overflow
encountered in double_scalars
  loss += reg*np.sum(W**2)
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/cs231n/classifiers/linear_svm.py:92: RuntimeWarning: overflow
encountered in square
  loss += reg*np.sum(W**2)

iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
iteration 0 / 1500: loss 1567.452984
iteration 100 / 1500: loss 43586140004327457813896229566026362613437234444136003
2141476856832276086745167421046043803633640996494662168572213197799424 0.000000
iteration 200 / 1500: loss 11255030471245350736830522361654013222970123491060179
66198425713198120688166967896650152955883695040344251316100848428106631968493499
80523915777602676738915233986193046608361817864421168557349312135108725348360059
431486071989657755897521661542 40.000000
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf

/content/cs231n/classifiers/linear_svm.py:113: RuntimeWarning: overflow
encountered in multiply
  dW += 2*reg*W
/content/cs231n/classifiers/linear_svm.py:87: RuntimeWarning: invalid value
encountered in matmul
  scores = X @ W
/content/cs231n/classifiers/linear_svm.py:89: RuntimeWarning: invalid value
encountered in less_equal
  margin[margin<=0] = 0
/content/cs231n/classifiers/linear_svm.py:108: RuntimeWarning: invalid value
encountered in greater
  margin[margin>0] = 1

iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 790.118482
iteration 100 / 1500: loss 712.471116
iteration 200 / 1500: loss 642.108452
iteration 300 / 1500: loss 579.841753
iteration 400 / 1500: loss 525.142099
iteration 500 / 1500: loss 475.549362
iteration 600 / 1500: loss 431.294922
iteration 700 / 1500: loss 391.217365
iteration 800 / 1500: loss 352.241369
iteration 900 / 1500: loss 318.481382
iteration 1000 / 1500: loss 288.601792
iteration 1100 / 1500: loss 261.371462
iteration 1200 / 1500: loss 236.519905
iteration 1300 / 1500: loss 214.261570
iteration 1400 / 1500: loss 194.870558
iteration 0 / 1500: loss 1546.196444
iteration 100 / 1500: loss 1261.211400
iteration 200 / 1500: loss 1032.296563
iteration 300 / 1500: loss 843.644228
iteration 400 / 1500: loss 691.932340
iteration 500 / 1500: loss 567.700913
iteration 600 / 1500: loss 464.719904
iteration 700 / 1500: loss 380.910513
iteration 800 / 1500: loss 312.696916
iteration 900 / 1500: loss 256.669096
iteration 1000 / 1500: loss 211.809884
iteration 1100 / 1500: loss 173.454963
iteration 1200 / 1500: loss 142.501323
iteration 1300 / 1500: loss 117.893094
iteration 1400 / 1500: loss 97.221679
iteration 0 / 1500: loss 790.572851
iteration 100 / 1500: loss inf

/content/cs231n/classifiers/linear_svm.py:88: RuntimeWarning: overflow
encountered in subtract
  margin = scores - scores[np.arange(num_train), y].reshape(-1,1) + 1
/content/cs231n/classifiers/linear_svm.py:87: RuntimeWarning: overflow
encountered in matmul
  scores = X @ W
/content/cs231n/classifiers/linear_svm.py:88: RuntimeWarning: invalid value
encountered in subtract
  margin = scores - scores[np.arange(num_train), y].reshape(-1,1) + 1

iteration 200 / 1500: loss nan
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 1529.158837
iteration 100 / 1500: loss inf
iteration 200 / 1500: loss nan
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 797.944692
iteration 100 / 1500: loss 479.302598
iteration 200 / 1500: loss 292.087456
iteration 300 / 1500: loss 177.026226
iteration 400 / 1500: loss 109.443664
iteration 500 / 1500: loss 67.522183
iteration 600 / 1500: loss 43.894327
iteration 700 / 1500: loss 28.222151
iteration 800 / 1500: loss 19.001534
iteration 900 / 1500: loss 13.541731
iteration 1000 / 1500: loss 10.853510
iteration 1100 / 1500: loss 8.414309
iteration 1200 / 1500: loss 6.809395
iteration 1300 / 1500: loss 6.503195
iteration 1400 / 1500: loss 6.149084
iteration 0 / 1500: loss 1576.363947
iteration 100 / 1500: loss 578.333289
iteration 200 / 1500: loss 214.263892
iteration 300 / 1500: loss 82.824541
iteration 400 / 1500: loss 33.267944
iteration 500 / 1500: loss 15.985398
iteration 600 / 1500: loss 10.056605
iteration 700 / 1500: loss 7.395112
iteration 800 / 1500: loss 5.894877
iteration 900 / 1500: loss 6.092817
iteration 1000 / 1500: loss 5.615112
iteration 1100 / 1500: loss 6.029179
iteration 1200 / 1500: loss 5.065956
iteration 1300 / 1500: loss 5.354368
iteration 1400 / 1500: loss 5.874967
iteration 0 / 1500: loss 787.057174
iteration 100 / 1500: loss 10.013377
iteration 200 / 1500: loss 5.615343
iteration 300 / 1500: loss 5.990488
iteration 400 / 1500: loss 5.701549
iteration 500 / 1500: loss 5.860040
iteration 600 / 1500: loss 6.065973
iteration 700 / 1500: loss 5.588820
iteration 800 / 1500: loss 6.483294
iteration 900 / 1500: loss 6.273596
iteration 1000 / 1500: loss 5.354767
iteration 1100 / 1500: loss 5.849633
iteration 1200 / 1500: loss 5.798406
iteration 1300 / 1500: loss 5.840293
iteration 1400 / 1500: loss 5.305015
iteration 0 / 1500: loss 1555.399000
iteration 100 / 1500: loss 5.934797
iteration 200 / 1500: loss 5.875517
iteration 300 / 1500: loss 6.321647
iteration 400 / 1500: loss 5.740718
iteration 500 / 1500: loss 6.311098
iteration 600 / 1500: loss 6.764900
iteration 700 / 1500: loss 6.101450
iteration 800 / 1500: loss 5.639432
iteration 900 / 1500: loss 5.873966
iteration 1000 / 1500: loss 6.563660
iteration 1100 / 1500: loss 6.070026
iteration 1200 / 1500: loss 5.959979
iteration 1300 / 1500: loss 6.595295
iteration 1400 / 1500: loss 6.660393
iteration 0 / 1500: loss 796.586946
iteration 100 / 1500: loss 126801914521490464321748051024283122975859886665076787
708819752368274319555994725387908203606702468075981140895703847395250746565204708
844061413662123792148205129750742797104489307911500758103903808261787640694780723
320.000000
iteration 200 / 1500: loss inf
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 1567.684586
iteration 100 / 1500: loss 18023671247389801287808987547574573669911033969352996
2548495027252491930898664860866660235964988843849288889027245186173189467378135
6401845499067074316792755756113834197711126720995889233503934181862378174580972
19241328194054585790104780429874738985270658883263905050521208094 72.000000
iteration 200 / 1500: loss inf
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 804.305877
iteration 100 / 1500: loss 621.361330
iteration 200 / 1500: loss 483.831940
iteration 300 / 1500: loss 376.574295
iteration 400 / 1500: loss 293.070602
iteration 500 / 1500: loss 229.901198
iteration 600 / 1500: loss 179.525189
iteration 700 / 1500: loss 140.038936
iteration 800 / 1500: loss 111.003949
iteration 900 / 1500: loss 86.725663
iteration 1000 / 1500: loss 68.827772
iteration 1100 / 1500: loss 53.945373
iteration 1200 / 1500: loss 43.201602
iteration 1300 / 1500: loss 35.297387
iteration 1400 / 1500: loss 28.777093
iteration 0 / 1500: loss 1546.895614
iteration 100 / 1500: loss 932.256855
iteration 200 / 1500: loss 566.490798
iteration 300 / 1500: loss 344.573644
iteration 400 / 1500: loss 210.691163
iteration 500 / 1500: loss 130.098471
iteration 600 / 1500: loss 80.723661
iteration 700 / 1500: loss 50.760753
iteration 800 / 1500: loss 32.705503
iteration 900 / 1500: loss 22.371593
iteration 1000 / 1500: loss 15.555648
iteration 1100 / 1500: loss 11.839469
iteration 1200 / 1500: loss 9.189074
iteration 1300 / 1500: loss 8.038724
iteration 1400 / 1500: loss 6.960774
iteration 0 / 1500: loss 784.079821
iteration 100 / 1500: loss 66.741295
iteration 200 / 1500: loss 9.887230
iteration 300 / 1500: loss 6.342129
iteration 400 / 1500: loss 4.915617
iteration 500 / 1500: loss 4.915608
iteration 600 / 1500: loss 4.697293
iteration 700 / 1500: loss 5.446924
iteration 800 / 1500: loss 5.833421
iteration 900 / 1500: loss 5.088804
iteration 1000 / 1500: loss 5.699817
iteration 1100 / 1500: loss 5.484531
iteration 1200 / 1500: loss 4.893771
iteration 1300 / 1500: loss 5.141491
iteration 1400 / 1500: loss 5.217196
iteration 0 / 1500: loss 1574.253437
iteration 100 / 1500: loss 15.616910
iteration 200 / 1500: loss 5.987566
iteration 300 / 1500: loss 5.806281
iteration 400 / 1500: loss 6.203602
iteration 500 / 1500: loss 5.685567
iteration 600 / 1500: loss 5.760319
iteration 700 / 1500: loss 5.532164
iteration 800 / 1500: loss 6.054227
iteration 900 / 1500: loss 5.683971
iteration 1000 / 1500: loss 6.142921
iteration 1100 / 1500: loss 6.557647
iteration 1200 / 1500: loss 5.540264
iteration 1300 / 1500: loss 5.915060
iteration 1400 / 1500: loss 5.568733
iteration 0 / 1500: loss 784.442835
iteration 100 / 1500: loss 783.015334
iteration 200 / 1500: loss 781.540332
iteration 300 / 1500: loss 782.798050
iteration 400 / 1500: loss 783.414736
iteration 500 / 1500: loss 777.772922
iteration 600 / 1500: loss 778.874703
iteration 700 / 1500: loss 781.182390
iteration 800 / 1500: loss 780.017895
iteration 900 / 1500: loss 776.062997
iteration 1000 / 1500: loss 778.201709
iteration 1100 / 1500: loss 775.243483
iteration 1200 / 1500: loss 776.541069
iteration 1300 / 1500: loss 772.818006
iteration 1400 / 1500: loss 771.765681
iteration 0 / 1500: loss 1566.666080
iteration 100 / 1500: loss 1567.330654
iteration 200 / 1500: loss 1562.872916
iteration 300 / 1500: loss 1559.441667
iteration 400 / 1500: loss 1556.662655
iteration 500 / 1500: loss 1551.589283
iteration 600 / 1500: loss 1552.521116
iteration 700 / 1500: loss 1545.122625
iteration 800 / 1500: loss 1543.796432
iteration 900 / 1500: loss 1538.162002
iteration 1000 / 1500: loss 1537.019682
iteration 1100 / 1500: loss 1534.846604
iteration 1200 / 1500: loss 1530.719344
iteration 1300 / 1500: loss 1527.666545
iteration 1400 / 1500: loss 1524.685888
iteration 0 / 1500: loss 783.613765
iteration 100 / 1500: loss 782.526486
iteration 200 / 1500: loss 784.083931
iteration 300 / 1500: loss 785.489750
iteration 400 / 1500: loss 784.522278
iteration 500 / 1500: loss 783.128985
iteration 600 / 1500: loss 782.624566
iteration 700 / 1500: loss 782.716403
iteration 800 / 1500: loss 782.136583
iteration 900 / 1500: loss 782.345474
iteration 1000 / 1500: loss 781.979078
iteration 1100 / 1500: loss 782.136173
iteration 1200 / 1500: loss 779.991916
iteration 1300 / 1500: loss 784.006451
iteration 1400 / 1500: loss 781.526458
iteration 0 / 1500: loss 1555.972091
iteration 100 / 1500: loss 1556.303310
iteration 200 / 1500: loss 1552.947369
iteration 300 / 1500: loss 1557.873310
iteration 400 / 1500: loss 1554.221352
iteration 500 / 1500: loss 1554.951687
iteration 600 / 1500: loss 1555.712628
iteration 700 / 1500: loss 1555.610443
iteration 800 / 1500: loss 1557.401891
iteration 900 / 1500: loss 1554.553106
iteration 1000 / 1500: loss 1553.703648
iteration 1100 / 1500: loss 1556.473274
iteration 1200 / 1500: loss 1552.774477
iteration 1300 / 1500: loss 1556.882961
iteration 1400 / 1500: loss 1554.122149
iteration 0 / 1500: loss 791.791544
iteration 100 / 1500: loss 780.639421
iteration 200 / 1500: loss 777.095914
iteration 300 / 1500: loss 764.596576
iteration 400 / 1500: loss 757.853934
iteration 500 / 1500: loss 749.825872
iteration 600 / 1500: loss 741.043453
iteration 700 / 1500: loss 731.308438
iteration 800 / 1500: loss 726.208111
iteration 900 / 1500: loss 720.080478
iteration 1000 / 1500: loss 709.943832
iteration 1100 / 1500: loss 705.737677
iteration 1200 / 1500: loss 696.397064
iteration 1300 / 1500: loss 689.060564
iteration 1400 / 1500: loss 679.194572
iteration 0 / 1500: loss 1550.880536
iteration 100 / 1500: loss 1521.241040
iteration 200 / 1500: loss 1490.060769
iteration 300 / 1500: loss 1462.520229
iteration 400 / 1500: loss 1430.812989
iteration 500 / 1500: loss 1403.306033
iteration 600 / 1500: loss 1374.115818
iteration 700 / 1500: loss 1347.895605
iteration 800 / 1500: loss 1318.258117
iteration 900 / 1500: loss 1293.640494
iteration 1000 / 1500: loss 1266.170812
iteration 1100 / 1500: loss 1241.910495
iteration 1200 / 1500: loss 1216.289953
iteration 1300 / 1500: loss 1193.677715
iteration 1400 / 1500: loss 1170.640464
lr 1.000000e-12 reg 2.500000e+04 train accuracy: 0.097898 val accuracy: 0.103000
lr 1.000000e-12 reg 5.000000e+04 train accuracy: 0.099510 val accuracy: 0.084000
lr 1.000000e-10 reg 2.500000e+04 train accuracy: 0.100429 val accuracy: 0.106000
lr 1.000000e-10 reg 5.000000e+04 train accuracy: 0.092306 val accuracy: 0.101000
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.156816 val accuracy: 0.158000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.150265 val accuracy: 0.149000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.267959 val accuracy: 0.291000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.306735 val accuracy: 0.323000
lr 2.500000e-08 reg 2.500000e+04 train accuracy: 0.350143 val accuracy: 0.350000
lr 2.500000e-08 reg 5.000000e+04 train accuracy: 0.359878 val accuracy: 0.373000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.369388 val accuracy: 0.378000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.359816 val accuracy: 0.362000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.366612 val accuracy: 0.370000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.354265 val accuracy: 0.363000
lr 2.500000e-07 reg 2.500000e+04 train accuracy: 0.355918 val accuracy: 0.359000
lr 2.500000e-07 reg 5.000000e+04 train accuracy: 0.337469 val accuracy: 0.342000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.349408 val accuracy: 0.352000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.310633 val accuracy: 0.326000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.053837 val accuracy: 0.058000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.500000e-04 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.500000e-04 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
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: 0.378000
```

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

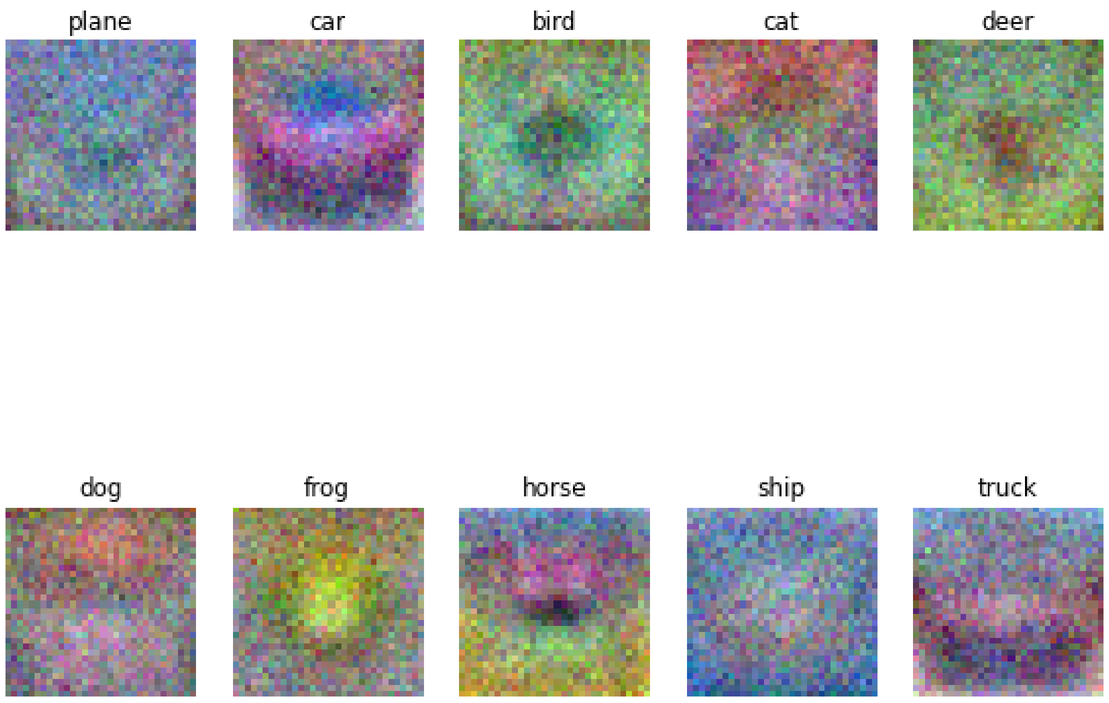


[17]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.364000
```

[18]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
```

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

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



Inline question 2

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

$Your Answer:$ The visualized weights look like averages of the images that are in each class. Any color bias means that the images in the class from the dataset contained mainly the dominant color. Many of these images seem to have flipped and rotated versions of a typical image in order to correctly classify any variations.

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.

2. Execute the cell below to download the modified .py files back to your drive.

[17]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/linear_svm.py', 'cs231n/classifiers/
↪linear_classifier.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
↪as f:
    f.write(''.join(open(files).readlines()))
```

softmax

September 12, 2020

[1]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'cs231n/assignment1/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&scope=email%20https%3a%2f%2fwww.googl
eapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%2
0https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2
fwww.googleapis.com%2fauth%2fpeopleapi.readonly&response_type=code

Enter your authorization code:
4/4AGCKkgf_sTs5PdHFIJuXHkDHLVpcDqTXT7NBx5BOYiHukAiVG59GGo
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-09-10 19:02:10--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
```

```
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  55.1MB/s    in 3.0s

2020-09-10 19:02:13 (55.1 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

1 Softmax exercise

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

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

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

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

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

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

```
%autoreload 2
```

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

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

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

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

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

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

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


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

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

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

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

from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

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

```
loss: 2.308863
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : The probability of each of our classes should be roughly the same. As there are 10 classes, we end up with a 1/10 probability, i.e. -log(0.1).

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

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

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

```
numerical: 1.023073 analytic: 1.023073, relative error: 1.157481e-08
numerical: -1.330196 analytic: -1.330196, relative error: 2.708230e-08
numerical: -2.918263 analytic: -2.918263, relative error: 1.174081e-08
numerical: -1.207850 analytic: -1.207850, relative error: 1.640992e-08
numerical: -0.154962 analytic: -0.154962, relative error: 2.039734e-07
numerical: -2.309255 analytic: -2.309255, relative error: 1.762565e-08
numerical: 1.285160 analytic: 1.285160, relative error: 7.947140e-08
numerical: -1.380554 analytic: -1.380554, relative error: 2.116940e-09
numerical: 1.016468 analytic: 1.016468, relative error: 3.289364e-08
numerical: -0.940313 analytic: -0.940313, relative error: 2.740769e-09
numerical: 2.425626 analytic: 2.425626, relative error: 2.228095e-08
numerical: -1.090587 analytic: -1.090587, relative error: 4.346658e-08
numerical: 0.427054 analytic: 0.427054, relative error: 8.848211e-08
numerical: 0.876338 analytic: 0.876338, relative error: 5.896106e-08
numerical: 0.340249 analytic: 0.340249, relative error: 6.266286e-08
numerical: 1.074447 analytic: 1.074447, relative error: 3.531512e-08
numerical: -2.537357 analytic: -2.537357, relative error: 1.793058e-08
numerical: 1.384497 analytic: 1.384497, relative error: 9.395274e-08
numerical: 1.318346 analytic: 1.318346, relative error: 4.239229e-08
numerical: -1.123714 analytic: -1.123714, relative error: 3.576198e-08
```

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

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

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

```
naive loss: 2.308863e+00 computed in 0.237042s
vectorized loss: 2.308863e+00 computed in 0.015977s
Loss difference: 0.000000
Gradient difference: 0.000000
```

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

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

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

```
# the best trained softmax classifer in best_softmax.                          ␣
↪#
################################################################################

# Provided as a reference. You may or may not want to change these␣
↪hyperparameters
learning_rates = [1e-7, 5e-7, 2.5e-7, 1e-8, 2.5e-8, 5e-8, 1e-9, 2.5e-9, 5e-9,␣
↪1e-5, 2.5e-5, 5e-5]
regularization_strengths = [2.5e4, 5e4, 1e4, 1e3]

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

for lr in learning_rates:
  for rs in regularization_strengths:
    sfm = Softmax()
    loss_hist = sfm.train(X_train, y_train, learning_rate = lr, reg = rs,␣
↪num_iters = 1500, verbose = True)
    train_pred = sfm.predict(X_train)
    val_pred = sfm.predict(X_val)
    train_acc = np.mean(train_pred == y_train)
    val_acc = np.mean(val_pred == y_val)
    results[(lr, rs)] = (train_acc, val_acc)
    if val_acc > best_val:
      best_val = val_acc
      best_softmax = sfm

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

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

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

```
iteration 0 / 1500: loss 767.933268
iteration 100 / 1500: loss 281.914511
iteration 200 / 1500: loss 104.485135
iteration 300 / 1500: loss 39.548676
iteration 400 / 1500: loss 15.760969
iteration 500 / 1500: loss 7.176367
iteration 600 / 1500: loss 3.868934
iteration 700 / 1500: loss 2.751380
iteration 800 / 1500: loss 2.349174
iteration 900 / 1500: loss 2.190564
iteration 1000 / 1500: loss 2.143839
iteration 1100 / 1500: loss 2.092634
iteration 1200 / 1500: loss 2.082417
iteration 1300 / 1500: loss 2.059906
iteration 1400 / 1500: loss 2.080999
iteration 0 / 1500: loss 1561.870287
iteration 100 / 1500: loss 210.366112
iteration 200 / 1500: loss 29.959841
iteration 300 / 1500: loss 5.829990
iteration 400 / 1500: loss 2.641983
iteration 500 / 1500: loss 2.147878
iteration 600 / 1500: loss 2.146312
iteration 700 / 1500: loss 2.120195
iteration 800 / 1500: loss 2.086777
iteration 900 / 1500: loss 2.089763
iteration 1000 / 1500: loss 2.174992
iteration 1100 / 1500: loss 2.157854
iteration 1200 / 1500: loss 2.134658
iteration 1300 / 1500: loss 2.167990
iteration 1400 / 1500: loss 2.093702
iteration 0 / 1500: loss 311.085086
iteration 100 / 1500: loss 208.033229
iteration 200 / 1500: loss 139.445510
iteration 300 / 1500: loss 93.863873
iteration 400 / 1500: loss 63.578054
iteration 500 / 1500: loss 43.109829
iteration 600 / 1500: loss 29.370568
iteration 700 / 1500: loss 20.381065
iteration 800 / 1500: loss 14.308375
iteration 900 / 1500: loss 10.175253
iteration 1000 / 1500: loss 7.478892
iteration 1100 / 1500: loss 5.700981
iteration 1200 / 1500: loss 4.543165
iteration 1300 / 1500: loss 3.632282
iteration 1400 / 1500: loss 3.149679
iteration 0 / 1500: loss 36.483786
iteration 100 / 1500: loss 33.673872
iteration 200 / 1500: loss 31.920264
iteration 300 / 1500: loss 30.435517
iteration 400 / 1500: loss 29.018620
iteration 500 / 1500: loss 27.738953
iteration 600 / 1500: loss 26.796788
iteration 700 / 1500: loss 25.554291
iteration 800 / 1500: loss 24.502395
iteration 900 / 1500: loss 23.681582
iteration 1000 / 1500: loss 22.736012
iteration 1100 / 1500: loss 21.960926
iteration 1200 / 1500: loss 21.099572
iteration 1300 / 1500: loss 20.487221
iteration 1400 / 1500: loss 19.404256
iteration 0 / 1500: loss 768.695463
iteration 100 / 1500: loss 6.907062
iteration 200 / 1500: loss 2.128554
iteration 300 / 1500: loss 2.084419
iteration 400 / 1500: loss 2.086628
iteration 500 / 1500: loss 2.098872
iteration 600 / 1500: loss 2.020452
iteration 700 / 1500: loss 2.112748
iteration 800 / 1500: loss 2.066751
iteration 900 / 1500: loss 2.059255
iteration 1000 / 1500: loss 2.086366
iteration 1100 / 1500: loss 2.137376
iteration 1200 / 1500: loss 2.108674
iteration 1300 / 1500: loss 2.091862
iteration 1400 / 1500: loss 2.082712
iteration 0 / 1500: loss 1537.985222
iteration 100 / 1500: loss 2.216863
iteration 200 / 1500: loss 2.149716
iteration 300 / 1500: loss 2.152309
iteration 400 / 1500: loss 2.112131
iteration 500 / 1500: loss 2.137405
iteration 600 / 1500: loss 2.145061
iteration 700 / 1500: loss 2.130132
iteration 800 / 1500: loss 2.142983
iteration 900 / 1500: loss 2.158718
iteration 1000 / 1500: loss 2.125700
iteration 1100 / 1500: loss 2.166035
iteration 1200 / 1500: loss 2.150093
iteration 1300 / 1500: loss 2.163754
iteration 1400 / 1500: loss 2.167505
iteration 0 / 1500: loss 315.776778
iteration 100 / 1500: loss 43.207841
iteration 200 / 1500: loss 7.560401
iteration 300 / 1500: loss 2.682483
iteration 400 / 1500: loss 2.135273
iteration 500 / 1500: loss 1.988386
iteration 600 / 1500: loss 2.021641
iteration 700 / 1500: loss 2.032326
iteration 800 / 1500: loss 1.968500
iteration 900 / 1500: loss 1.975336
iteration 1000 / 1500: loss 2.002844
iteration 1100 / 1500: loss 1.940676
iteration 1200 / 1500: loss 1.959892
iteration 1300 / 1500: loss 1.969253
iteration 1400 / 1500: loss 2.074332
iteration 0 / 1500: loss 36.202587
iteration 100 / 1500: loss 27.643408
iteration 200 / 1500: loss 22.778400
iteration 300 / 1500: loss 18.845673
iteration 400 / 1500: loss 15.525906
iteration 500 / 1500: loss 13.053575
iteration 600 / 1500: loss 11.008502
iteration 700 / 1500: loss 9.247539
iteration 800 / 1500: loss 7.794430
iteration 900 / 1500: loss 6.852312
iteration 1000 / 1500: loss 5.708107
iteration 1100 / 1500: loss 5.120283
iteration 1200 / 1500: loss 4.450788
iteration 1300 / 1500: loss 4.002504
iteration 1400 / 1500: loss 3.664493
iteration 0 / 1500: loss 781.336097
iteration 100 / 1500: loss 64.463535
iteration 200 / 1500: loss 7.117875
iteration 300 / 1500: loss 2.499383
iteration 400 / 1500: loss 2.117772
iteration 500 / 1500: loss 2.139047
iteration 600 / 1500: loss 2.064846
iteration 700 / 1500: loss 2.043922
iteration 800 / 1500: loss 2.045811
iteration 900 / 1500: loss 2.072484
iteration 1000 / 1500: loss 2.161231
iteration 1100 / 1500: loss 2.026522
iteration 1200 / 1500: loss 2.029901
iteration 1300 / 1500: loss 2.068132
iteration 1400 / 1500: loss 2.059744
iteration 0 / 1500: loss 1554.665296
iteration 100 / 1500: loss 11.896764
iteration 200 / 1500: loss 2.191059
iteration 300 / 1500: loss 2.148780
iteration 400 / 1500: loss 2.137381
iteration 500 / 1500: loss 2.154577
iteration 600 / 1500: loss 2.117759
iteration 700 / 1500: loss 2.131659
iteration 800 / 1500: loss 2.088159
iteration 900 / 1500: loss 2.116902
iteration 1000 / 1500: loss 2.143115
iteration 1100 / 1500: loss 2.158903
iteration 1200 / 1500: loss 2.139089
iteration 1300 / 1500: loss 2.153860
iteration 1400 / 1500: loss 2.135095
iteration 0 / 1500: loss 312.100450
iteration 100 / 1500: loss 114.624758
iteration 200 / 1500: loss 43.208479
iteration 300 / 1500: loss 16.960348
iteration 400 / 1500: loss 7.429727
iteration 500 / 1500: loss 3.989206
iteration 600 / 1500: loss 2.659624
iteration 700 / 1500: loss 2.285675
iteration 800 / 1500: loss 2.093562
iteration 900 / 1500: loss 2.002243
iteration 1000 / 1500: loss 2.014338
iteration 1100 / 1500: loss 2.029217
iteration 1200 / 1500: loss 1.979935
iteration 1300 / 1500: loss 2.086170
iteration 1400 / 1500: loss 2.060492
iteration 0 / 1500: loss 35.668845
iteration 100 / 1500: loss 30.944779
iteration 200 / 1500: loss 27.942378
iteration 300 / 1500: loss 25.193300
iteration 400 / 1500: loss 22.756912
iteration 500 / 1500: loss 20.649994
iteration 600 / 1500: loss 18.642198
iteration 700 / 1500: loss 17.141132
iteration 800 / 1500: loss 15.672707
iteration 900 / 1500: loss 14.223318
iteration 1000 / 1500: loss 13.064025
iteration 1100 / 1500: loss 12.174615
iteration 1200 / 1500: loss 10.830565
iteration 1300 / 1500: loss 10.043628
iteration 1400 / 1500: loss 9.233736
iteration 0 / 1500: loss 771.208290
iteration 100 / 1500: loss 697.844084
iteration 200 / 1500: loss 631.269646
iteration 300 / 1500: loss 571.108593
iteration 400 / 1500: loss 516.806674
iteration 500 / 1500: loss 467.958990
iteration 600 / 1500: loss 423.137028
iteration 700 / 1500: loss 382.757982
iteration 800 / 1500: loss 346.712151
iteration 900 / 1500: loss 313.536770
iteration 1000 / 1500: loss 283.804014
iteration 1100 / 1500: loss 256.714548
iteration 1200 / 1500: loss 232.546569
iteration 1300 / 1500: loss 210.680009
iteration 1400 / 1500: loss 190.569688
iteration 0 / 1500: loss 1526.539668
iteration 100 / 1500: loss 1250.057100
iteration 200 / 1500: loss 1023.430264
iteration 300 / 1500: loss 837.858441
iteration 400 / 1500: loss 685.950972
iteration 500 / 1500: loss 561.753520
iteration 600 / 1500: loss 460.173013
iteration 700 / 1500: loss 377.040369
iteration 800 / 1500: loss 308.897751
iteration 900 / 1500: loss 253.170164
iteration 1000 / 1500: loss 207.665479
iteration 1100 / 1500: loss 170.306684
iteration 1200 / 1500: loss 139.793948
iteration 1300 / 1500: loss 114.691732
iteration 1400 / 1500: loss 94.216553
iteration 0 / 1500: loss 314.919134
iteration 100 / 1500: loss 302.262522
iteration 200 / 1500: loss 290.383240
iteration 300 / 1500: loss 278.603916
iteration 400 / 1500: loss 267.687913
iteration 500 / 1500: loss 256.854837
iteration 600 / 1500: loss 246.548836
iteration 700 / 1500: loss 236.874261
iteration 800 / 1500: loss 227.567052
iteration 900 / 1500: loss 218.764296
iteration 1000 / 1500: loss 210.271725
iteration 1100 / 1500: loss 201.767032
iteration 1200 / 1500: loss 194.189762
iteration 1300 / 1500: loss 186.610908
iteration 1400 / 1500: loss 178.798856
iteration 0 / 1500: loss 37.305486
iteration 100 / 1500: loss 36.880469
iteration 200 / 1500: loss 35.963723
iteration 300 / 1500: loss 35.560710
iteration 400 / 1500: loss 35.167517
iteration 500 / 1500: loss 35.142287
iteration 600 / 1500: loss 34.481638
iteration 700 / 1500: loss 34.449095
iteration 800 / 1500: loss 34.448381
iteration 900 / 1500: loss 33.806412
iteration 1000 / 1500: loss 33.896029
iteration 1100 / 1500: loss 33.300604
iteration 1200 / 1500: loss 33.357920
iteration 1300 / 1500: loss 33.224396
iteration 1400 / 1500: loss 33.007055
iteration 0 / 1500: loss 776.194876
iteration 100 / 1500: loss 604.259707
iteration 200 / 1500: loss 470.625830
iteration 300 / 1500: loss 366.362140
iteration 400 / 1500: loss 285.509985
iteration 500 / 1500: loss 222.614175
iteration 600 / 1500: loss 173.735903
iteration 700 / 1500: loss 135.697976
iteration 800 / 1500: loss 105.976642
iteration 900 / 1500: loss 82.919919
iteration 1000 / 1500: loss 64.967739
iteration 1100 / 1500: loss 50.978989
iteration 1200 / 1500: loss 40.173679
iteration 1300 / 1500: loss 31.661349
iteration 1400 / 1500: loss 25.139828
iteration 0 / 1500: loss 1551.149360
iteration 100 / 1500: loss 940.148101
iteration 200 / 1500: loss 570.195812
iteration 300 / 1500: loss 346.138838
iteration 400 / 1500: loss 210.424474
iteration 500 / 1500: loss 128.428983
iteration 600 / 1500: loss 78.622414
iteration 700 / 1500: loss 48.413120
iteration 800 / 1500: loss 30.184719
iteration 900 / 1500: loss 19.143424
iteration 1000 / 1500: loss 12.384368
iteration 1100 / 1500: loss 8.358342
iteration 1200 / 1500: loss 5.918609
iteration 1300 / 1500: loss 4.402274
iteration 1400 / 1500: loss 3.538090
iteration 0 / 1500: loss 311.122923
iteration 100 / 1500: loss 280.978769
iteration 200 / 1500: loss 254.409243
iteration 300 / 1500: loss 229.856371
iteration 400 / 1500: loss 208.248211
iteration 500 / 1500: loss 187.920857
iteration 600 / 1500: loss 170.315627
iteration 700 / 1500: loss 154.102761
iteration 800 / 1500: loss 139.510646
iteration 900 / 1500: loss 126.391229
iteration 1000 / 1500: loss 114.615034
iteration 1100 / 1500: loss 103.669789
iteration 1200 / 1500: loss 93.993484
iteration 1300 / 1500: loss 85.053856
iteration 1400 / 1500: loss 77.059263
iteration 0 / 1500: loss 36.575865
iteration 100 / 1500: loss 35.531448
iteration 200 / 1500: loss 35.067086
iteration 300 / 1500: loss 34.319203
iteration 400 / 1500: loss 33.974122
iteration 500 / 1500: loss 33.600498
iteration 600 / 1500: loss 33.013709
iteration 700 / 1500: loss 32.957180
iteration 800 / 1500: loss 32.284952
iteration 900 / 1500: loss 31.805505
iteration 1000 / 1500: loss 31.275698
iteration 1100 / 1500: loss 31.111719
iteration 1200 / 1500: loss 30.704238
iteration 1300 / 1500: loss 30.586461
iteration 1400 / 1500: loss 29.968800
iteration 0 / 1500: loss 780.221577
iteration 100 / 1500: loss 473.029074
iteration 200 / 1500: loss 286.846459
iteration 300 / 1500: loss 174.512925
iteration 400 / 1500: loss 106.457002
iteration 500 / 1500: loss 65.199865
iteration 600 / 1500: loss 40.289554
iteration 700 / 1500: loss 25.219044
iteration 800 / 1500: loss 16.099546
iteration 900 / 1500: loss 10.534685
iteration 1000 / 1500: loss 7.308000
iteration 1100 / 1500: loss 5.206986
iteration 1200 / 1500: loss 3.996072
iteration 1300 / 1500: loss 3.229179
iteration 1400 / 1500: loss 2.808288
iteration 0 / 1500: loss 1525.714835
iteration 100 / 1500: loss 559.824774
iteration 200 / 1500: loss 206.438299
iteration 300 / 1500: loss 76.956386
iteration 400 / 1500: loss 29.555366
iteration 500 / 1500: loss 12.212558
iteration 600 / 1500: loss 5.835782
iteration 700 / 1500: loss 3.512660
iteration 800 / 1500: loss 2.660527
iteration 900 / 1500: loss 2.285476
iteration 1000 / 1500: loss 2.223811
iteration 1100 / 1500: loss 2.211191
iteration 1200 / 1500: loss 2.144452
iteration 1300 / 1500: loss 2.105251
iteration 1400 / 1500: loss 2.157666
iteration 0 / 1500: loss 314.510245
iteration 100 / 1500: loss 256.407989
iteration 200 / 1500: loss 209.740507
iteration 300 / 1500: loss 171.673578
iteration 400 / 1500: loss 140.593118
iteration 500 / 1500: loss 115.295116
iteration 600 / 1500: loss 94.521633
iteration 700 / 1500: loss 77.698108
iteration 800 / 1500: loss 63.808787
iteration 900 / 1500: loss 52.637646
iteration 1000 / 1500: loss 43.393851
iteration 1100 / 1500: loss 35.757437
iteration 1200 / 1500: loss 29.578209
iteration 1300 / 1500: loss 24.506492
iteration 1400 / 1500: loss 20.612283
iteration 0 / 1500: loss 36.487881
iteration 100 / 1500: loss 34.149939
iteration 200 / 1500: loss 33.242679
iteration 300 / 1500: loss 32.118424
iteration 400 / 1500: loss 31.530945
iteration 500 / 1500: loss 30.894568
iteration 600 / 1500: loss 30.291794
iteration 700 / 1500: loss 29.456026
iteration 800 / 1500: loss 28.952986
iteration 900 / 1500: loss 28.138555
iteration 1000 / 1500: loss 27.604322
iteration 1100 / 1500: loss 27.090480
iteration 1200 / 1500: loss 26.541338
iteration 1300 / 1500: loss 26.162184
iteration 1400 / 1500: loss 25.245383
iteration 0 / 1500: loss 781.985257
iteration 100 / 1500: loss 773.973641
iteration 200 / 1500: loss 766.459408
iteration 300 / 1500: loss 759.062392
iteration 400 / 1500: loss 751.167780
iteration 500 / 1500: loss 743.787665
iteration 600 / 1500: loss 736.567681
iteration 700 / 1500: loss 728.998809
iteration 800 / 1500: loss 721.591381
iteration 900 / 1500: loss 714.797623
iteration 1000 / 1500: loss 707.831845
iteration 1100 / 1500: loss 700.012935
iteration 1200 / 1500: loss 693.280341
iteration 1300 / 1500: loss 686.162915
iteration 1400 / 1500: loss 679.626125
iteration 0 / 1500: loss 1559.602377
iteration 100 / 1500: loss 1528.243118
iteration 200 / 1500: loss 1497.443036
iteration 300 / 1500: loss 1468.123031
iteration 400 / 1500: loss 1439.100057
iteration 500 / 1500: loss 1410.508932
iteration 600 / 1500: loss 1382.291701
iteration 700 / 1500: loss 1355.231478
iteration 800 / 1500: loss 1328.675768
iteration 900 / 1500: loss 1302.125652
iteration 1000 / 1500: loss 1276.028074
iteration 1100 / 1500: loss 1250.934636
iteration 1200 / 1500: loss 1225.984913
iteration 1300 / 1500: loss 1202.160432
iteration 1400 / 1500: loss 1178.519065
iteration 0 / 1500: loss 309.194118
iteration 100 / 1500: loss 307.287135
iteration 200 / 1500: loss 306.064078
iteration 300 / 1500: loss 304.947571
iteration 400 / 1500: loss 303.761033
iteration 500 / 1500: loss 301.497760
iteration 600 / 1500: loss 300.538384
iteration 700 / 1500: loss 299.482686
iteration 800 / 1500: loss 298.585209
iteration 900 / 1500: loss 297.112109
iteration 1000 / 1500: loss 295.745123
iteration 1100 / 1500: loss 294.078533
iteration 1200 / 1500: loss 293.204471
iteration 1300 / 1500: loss 291.873159
iteration 1400 / 1500: loss 291.123941
iteration 0 / 1500: loss 36.465482
iteration 100 / 1500: loss 36.664849
iteration 200 / 1500: loss 36.854769
iteration 300 / 1500: loss 36.593702
iteration 400 / 1500: loss 36.543090
iteration 500 / 1500: loss 36.340591
iteration 600 / 1500: loss 36.342613
iteration 700 / 1500: loss 36.301464
iteration 800 / 1500: loss 36.257397
iteration 900 / 1500: loss 36.103045
iteration 1000 / 1500: loss 35.947811
iteration 1100 / 1500: loss 36.098919
iteration 1200 / 1500: loss 36.092491
iteration 1300 / 1500: loss 36.084089
iteration 1400 / 1500: loss 35.856312
iteration 0 / 1500: loss 759.417893
iteration 100 / 1500: loss 740.769085
iteration 200 / 1500: loss 722.528234
iteration 300 / 1500: loss 704.318847
iteration 400 / 1500: loss 686.777577
iteration 500 / 1500: loss 669.710347
iteration 600 / 1500: loss 653.490817
iteration 700 / 1500: loss 637.298063
iteration 800 / 1500: loss 621.508237
iteration 900 / 1500: loss 606.446449
iteration 1000 / 1500: loss 590.939378
iteration 1100 / 1500: loss 576.251879
iteration 1200 / 1500: loss 562.356810
iteration 1300 / 1500: loss 548.219877
iteration 1400 / 1500: loss 534.827642
iteration 0 / 1500: loss 1545.601956
iteration 100 / 1500: loss 1470.742772
iteration 200 / 1500: loss 1398.304296
iteration 300 / 1500: loss 1330.574983
iteration 400 / 1500: loss 1265.580975
iteration 500 / 1500: loss 1203.893203
iteration 600 / 1500: loss 1144.554433
iteration 700 / 1500: loss 1089.188338
iteration 800 / 1500: loss 1036.358426
iteration 900 / 1500: loss 985.547804
iteration 1000 / 1500: loss 937.540551
iteration 1100 / 1500: loss 891.764701
iteration 1200 / 1500: loss 848.273835
iteration 1300 / 1500: loss 807.032748
iteration 1400 / 1500: loss 767.806779
iteration 0 / 1500: loss 312.592608
iteration 100 / 1500: loss 309.231353
iteration 200 / 1500: loss 305.509996
iteration 300 / 1500: loss 302.530987
iteration 400 / 1500: loss 300.061773
iteration 500 / 1500: loss 297.064831
iteration 600 / 1500: loss 293.934935
iteration 700 / 1500: loss 290.842812
iteration 800 / 1500: loss 287.901422
iteration 900 / 1500: loss 285.134764
iteration 1000 / 1500: loss 282.162022
iteration 1100 / 1500: loss 279.251012
iteration 1200 / 1500: loss 276.204283
iteration 1300 / 1500: loss 273.762001
iteration 1400 / 1500: loss 271.145988
iteration 0 / 1500: loss 36.499574
iteration 100 / 1500: loss 35.844580
iteration 200 / 1500: loss 36.248675
iteration 300 / 1500: loss 36.181963
iteration 400 / 1500: loss 35.166559
iteration 500 / 1500: loss 35.217931
iteration 600 / 1500: loss 35.404953
iteration 700 / 1500: loss 35.374538
iteration 800 / 1500: loss 35.043593
iteration 900 / 1500: loss 35.200924
iteration 1000 / 1500: loss 35.144279
iteration 1100 / 1500: loss 34.810554
iteration 1200 / 1500: loss 34.861289
iteration 1300 / 1500: loss 34.636063
iteration 1400 / 1500: loss 34.344513
iteration 0 / 1500: loss 776.516318
iteration 100 / 1500: loss 738.581493
iteration 200 / 1500: loss 702.071660
iteration 300 / 1500: loss 667.947898
iteration 400 / 1500: loss 635.442537
iteration 500 / 1500: loss 604.663537
iteration 600 / 1500: loss 575.206955
iteration 700 / 1500: loss 546.798404
iteration 800 / 1500: loss 520.355826
iteration 900 / 1500: loss 495.114616
iteration 1000 / 1500: loss 470.848553
iteration 1100 / 1500: loss 447.978487
iteration 1200 / 1500: loss 426.008922
iteration 1300 / 1500: loss 405.235949
iteration 1400 / 1500: loss 385.428632
iteration 0 / 1500: loss 1539.247930
iteration 100 / 1500: loss 1392.476816
iteration 200 / 1500: loss 1259.729442
iteration 300 / 1500: loss 1139.862451
iteration 400 / 1500: loss 1031.364054
iteration 500 / 1500: loss 933.084014
iteration 600 / 1500: loss 844.621362
iteration 700 / 1500: loss 764.239553
iteration 800 / 1500: loss 691.869511
iteration 900 / 1500: loss 625.857763
iteration 1000 / 1500: loss 566.473944
iteration 1100 / 1500: loss 512.666954
iteration 1200 / 1500: loss 463.948203
iteration 1300 / 1500: loss 420.068472
iteration 1400 / 1500: loss 380.016508
iteration 0 / 1500: loss 309.338650
iteration 100 / 1500: loss 302.755368
iteration 200 / 1500: loss 297.113996
iteration 300 / 1500: loss 291.339864
iteration 400 / 1500: loss 285.144379
iteration 500 / 1500: loss 279.527431
iteration 600 / 1500: loss 273.621960
iteration 700 / 1500: loss 268.478504
iteration 800 / 1500: loss 263.407912
iteration 900 / 1500: loss 257.701357
iteration 1000 / 1500: loss 252.603169
iteration 1100 / 1500: loss 247.486468
iteration 1200 / 1500: loss 242.444207
iteration 1300 / 1500: loss 238.047665
iteration 1400 / 1500: loss 233.305255
iteration 0 / 1500: loss 36.604184
iteration 100 / 1500: loss 36.256577
iteration 200 / 1500: loss 35.436634
iteration 300 / 1500: loss 35.274514
iteration 400 / 1500: loss 34.997699
iteration 500 / 1500: loss 35.180929
iteration 600 / 1500: loss 34.709843
iteration 700 / 1500: loss 35.126713
iteration 800 / 1500: loss 34.441418
iteration 900 / 1500: loss 34.336568
iteration 1000 / 1500: loss 34.481600
iteration 1100 / 1500: loss 34.054098
iteration 1200 / 1500: loss 34.001960
iteration 1300 / 1500: loss 34.240385
iteration 1400 / 1500: loss 33.779434
iteration 0 / 1500: loss 778.743840
iteration 100 / 1500: loss 8.347660
iteration 200 / 1500: loss 7.771139
iteration 300 / 1500: loss 7.279537
iteration 400 / 1500: loss 9.635578
iteration 500 / 1500: loss 6.941875
iteration 600 / 1500: loss 7.169723
iteration 700 / 1500: loss 5.365003
iteration 800 / 1500: loss 8.192037
iteration 900 / 1500: loss 6.796105
iteration 1000 / 1500: loss 8.742610
iteration 1100 / 1500: loss 6.258474
iteration 1200 / 1500: loss 8.099143
iteration 1300 / 1500: loss 11.632836
iteration 1400 / 1500: loss 6.820494
iteration 0 / 1500: loss 1546.384011
iteration 100 / 1500: loss 16.843559
iteration 200 / 1500: loss 14.868470
iteration 300 / 1500: loss 14.297655
iteration 400 / 1500: loss 15.135243
iteration 500 / 1500: loss 15.685156
iteration 600 / 1500: loss 16.220968
iteration 700 / 1500: loss 18.304495
iteration 800 / 1500: loss 16.337105
iteration 900 / 1500: loss 16.181490
iteration 1000 / 1500: loss 15.520157
iteration 1100 / 1500: loss 17.892499
iteration 1200 / 1500: loss 14.553211
iteration 1300 / 1500: loss 17.364059
iteration 1400 / 1500: loss 17.762644
iteration 0 / 1500: loss 310.324847
iteration 100 / 1500: loss 6.296984
iteration 200 / 1500: loss 5.453700
iteration 300 / 1500: loss 4.815076
iteration 400 / 1500: loss 4.201912
iteration 500 / 1500: loss 5.277731
iteration 600 / 1500: loss 5.731131
iteration 700 / 1500: loss 5.718394
iteration 800 / 1500: loss 5.937434
iteration 900 / 1500: loss 5.630224
iteration 1000 / 1500: loss 4.999063
iteration 1100 / 1500: loss 5.546187
iteration 1200 / 1500: loss 7.765379
iteration 1300 / 1500: loss 4.868993
iteration 1400 / 1500: loss 5.375590
iteration 0 / 1500: loss 36.856301
iteration 100 / 1500: loss 4.082333
iteration 200 / 1500: loss 3.513992
iteration 300 / 1500: loss 3.870617
iteration 400 / 1500: loss 3.717952
iteration 500 / 1500: loss 2.739518
iteration 600 / 1500: loss 2.763917
iteration 700 / 1500: loss 2.738042
iteration 800 / 1500: loss 4.077956
iteration 900 / 1500: loss 3.093825
iteration 1000 / 1500: loss 3.392292
iteration 1100 / 1500: loss 2.993548
iteration 1200 / 1500: loss 3.495166
iteration 1300 / 1500: loss 3.945044
iteration 1400 / 1500: loss 3.074127
iteration 0 / 1500: loss 783.899113
iteration 100 / 1500: loss 68.208013
iteration 200 / 1500: loss 68.908849
iteration 300 / 1500: loss 81.378193
iteration 400 / 1500: loss 65.414709
iteration 500 / 1500: loss 65.863303
iteration 600 / 1500: loss 76.947600
iteration 700 / 1500: loss 63.829541
iteration 800 / 1500: loss 73.691970
iteration 900 / 1500: loss 64.067836
iteration 1000 / 1500: loss 67.721085
iteration 1100 / 1500: loss 73.986786
iteration 1200 / 1500: loss 72.339559
iteration 1300 / 1500: loss 68.917049
iteration 1400 / 1500: loss 74.164054
iteration 0 / 1500: loss 1539.798593

/content/cs231n/classifiers/softmax.py:79: RuntimeWarning: divide by zero
encountered in log
  loss = np.sum(-np.log(np.exp(f[range(nt),y])/total))
/content/cs231n/classifiers/softmax.py:79: RuntimeWarning: invalid value
encountered in true_divide
  loss = np.sum(-np.log(np.exp(f[range(nt),y])/total))
/content/cs231n/classifiers/softmax.py:83: RuntimeWarning: invalid value
encountered in true_divide
  tmp = np.exp(f) / total.reshape(-1,1)

iteration 100 / 1500: loss nan
iteration 200 / 1500: loss nan
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 312.437922
iteration 100 / 1500: loss 17.465845
iteration 200 / 1500: loss 23.342039
iteration 300 / 1500: loss 15.689672
iteration 400 / 1500: loss 24.536986
iteration 500 / 1500: loss 20.105711
iteration 600 / 1500: loss 20.598931
iteration 700 / 1500: loss 21.834160
iteration 800 / 1500: loss 21.485238
iteration 900 / 1500: loss 18.518624
iteration 1000 / 1500: loss 19.240594
iteration 1100 / 1500: loss 15.885719
iteration 1200 / 1500: loss 13.772597
iteration 1300 / 1500: loss 15.112940
iteration 1400 / 1500: loss 21.288051
iteration 0 / 1500: loss 36.592694
iteration 100 / 1500: loss 8.229626
iteration 200 / 1500: loss 11.625272
iteration 300 / 1500: loss 10.137036
iteration 400 / 1500: loss 6.257118
iteration 500 / 1500: loss 8.073441
iteration 600 / 1500: loss 8.547442
iteration 700 / 1500: loss 9.371661
iteration 800 / 1500: loss 11.109709
iteration 900 / 1500: loss 7.437829
iteration 1000 / 1500: loss 10.839104
iteration 1100 / 1500: loss 10.488949
iteration 1200 / 1500: loss 11.239757
iteration 1300 / 1500: loss 12.955167
iteration 1400 / 1500: loss 9.056056
iteration 0 / 1500: loss 772.425428
iteration 100 / 1500: loss nan
iteration 200 / 1500: loss nan
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 1536.254203
iteration 100 / 1500: loss nan
iteration 200 / 1500: loss nan
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
iteration 0 / 1500: loss 308.490539
iteration 100 / 1500: loss 101.507409
iteration 200 / 1500: loss 86.039584
iteration 300 / 1500: loss 85.144870
iteration 400 / 1500: loss 88.886329
iteration 500 / 1500: loss 88.310084
iteration 600 / 1500: loss 85.797278
iteration 700 / 1500: loss 87.183615
iteration 800 / 1500: loss 88.975663
iteration 900 / 1500: loss 85.233876
iteration 1000 / 1500: loss 84.677359
iteration 1100 / 1500: loss 95.513556
iteration 1200 / 1500: loss 86.798541
iteration 1300 / 1500: loss 100.605255
iteration 1400 / 1500: loss 88.450059
iteration 0 / 1500: loss 36.250728
iteration 100 / 1500: loss 18.134633
iteration 200 / 1500: loss 20.963508
iteration 300 / 1500: loss 25.545822
iteration 400 / 1500: loss 21.465395
iteration 500 / 1500: loss 22.330692
iteration 600 / 1500: loss 26.813503
iteration 700 / 1500: loss 15.084890
iteration 800 / 1500: loss 28.339153
iteration 900 / 1500: loss 19.326551
iteration 1000 / 1500: loss 20.381460
iteration 1100 / 1500: loss 19.862334
iteration 1200 / 1500: loss 16.124477
iteration 1300 / 1500: loss 21.847675
iteration 1400 / 1500: loss 22.183353
lr 1.000000e-09 reg 1.000000e+03 train accuracy: 0.076755 val accuracy: 0.080000
lr 1.000000e-09 reg 1.000000e+04 train accuracy: 0.071224 val accuracy: 0.063000
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.109020 val accuracy: 0.107000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.114041 val accuracy: 0.100000
lr 2.500000e-09 reg 1.000000e+03 train accuracy: 0.146469 val accuracy: 0.144000
lr 2.500000e-09 reg 1.000000e+04 train accuracy: 0.096878 val accuracy: 0.104000
lr 2.500000e-09 reg 2.500000e+04 train accuracy: 0.123694 val accuracy: 0.131000
lr 2.500000e-09 reg 5.000000e+04 train accuracy: 0.113612 val accuracy: 0.110000
lr 5.000000e-09 reg 1.000000e+03 train accuracy: 0.136510 val accuracy: 0.125000
lr 5.000000e-09 reg 1.000000e+04 train accuracy: 0.127918 val accuracy: 0.118000
lr 5.000000e-09 reg 2.500000e+04 train accuracy: 0.146980 val accuracy: 0.134000
lr 5.000000e-09 reg 5.000000e+04 train accuracy: 0.160714 val accuracy: 0.154000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.164082 val accuracy: 0.152000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.160265 val accuracy: 0.149000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.171571 val accuracy: 0.185000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.191673 val accuracy: 0.189000
lr 2.500000e-08 reg 1.000000e+03 train accuracy: 0.188898 val accuracy: 0.189000
lr 2.500000e-08 reg 1.000000e+04 train accuracy: 0.223306 val accuracy: 0.216000
lr 2.500000e-08 reg 2.500000e+04 train accuracy: 0.279694 val accuracy: 0.292000
lr 2.500000e-08 reg 5.000000e+04 train accuracy: 0.307816 val accuracy: 0.325000
lr 5.000000e-08 reg 1.000000e+03 train accuracy: 0.217061 val accuracy: 0.230000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.296204 val accuracy: 0.303000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.329837 val accuracy: 0.346000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.299612 val accuracy: 0.325000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.258286 val accuracy: 0.288000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.353163 val accuracy: 0.378000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.330102 val accuracy: 0.344000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.310163 val accuracy: 0.322000
lr 2.500000e-07 reg 1.000000e+03 train accuracy: 0.335184 val accuracy: 0.344000
lr 2.500000e-07 reg 1.000000e+04 train accuracy: 0.352776 val accuracy: 0.368000
lr 2.500000e-07 reg 2.500000e+04 train accuracy: 0.325816 val accuracy: 0.332000
lr 2.500000e-07 reg 5.000000e+04 train accuracy: 0.309082 val accuracy: 0.318000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.388265 val accuracy: 0.402000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.355327 val accuracy: 0.370000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.333612 val accuracy: 0.341000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.299408 val accuracy: 0.311000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.256102 val accuracy: 0.263000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.180327 val accuracy: 0.181000
lr 1.000000e-05 reg 2.500000e+04 train accuracy: 0.171388 val accuracy: 0.187000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.122878 val accuracy: 0.110000
lr 2.500000e-05 reg 1.000000e+03 train accuracy: 0.213918 val accuracy: 0.213000
lr 2.500000e-05 reg 1.000000e+04 train accuracy: 0.115612 val accuracy: 0.122000
lr 2.500000e-05 reg 2.500000e+04 train accuracy: 0.080143 val accuracy: 0.072000
lr 2.500000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 1.000000e+03 train accuracy: 0.173265 val accuracy: 0.197000
lr 5.000000e-05 reg 1.000000e+04 train accuracy: 0.131653 val accuracy: 0.137000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.402000
```

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

```
softmax on raw pixels final test set accuracy: 0.366000
```

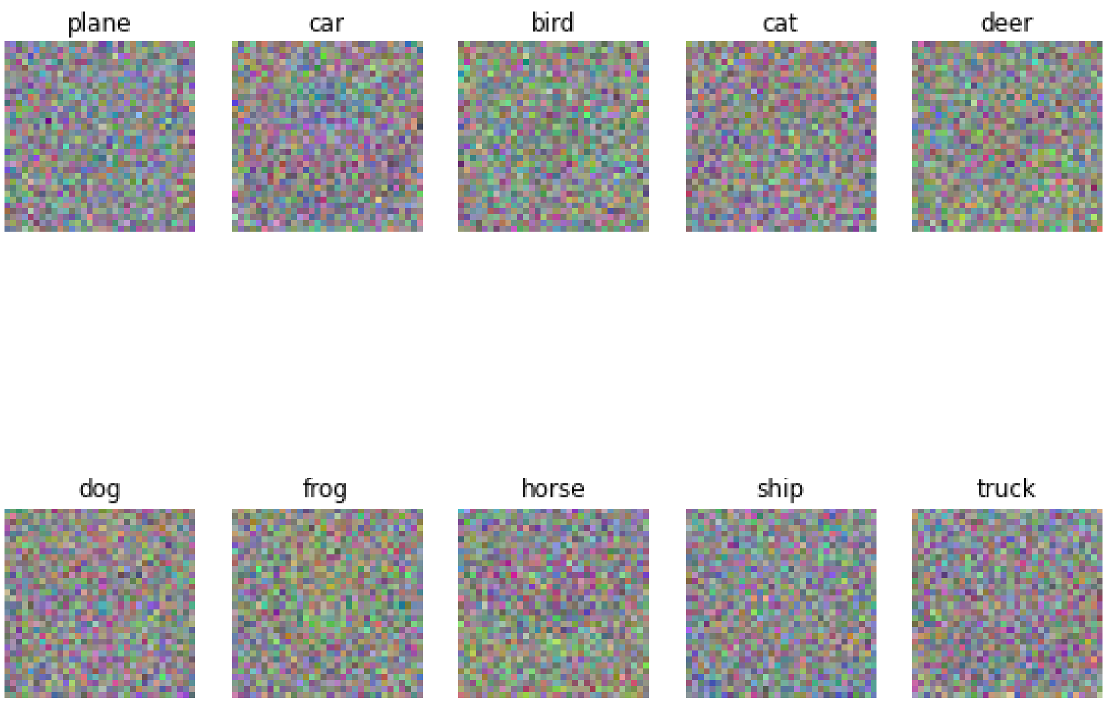
Inline Question 2 - *True or False*

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

YourAnswer : True

YourExplanation : The SVM Loss could be unchanged if the max expression results in 0 everytime. However, the softmax classifier cannot be 0 because the numerator will be less than the denominator inside of the log, meaning that it cannot be 0.

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

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

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

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



---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[10]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/softmax.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```

two_layer_net

September 12, 2020

[2]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'cs231n/assignment1/cs231n'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-09-12 15:12:35--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  13.3MB/s    in 14s

2020-09-12 15:12:50 (11.8 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]
```

```
cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

[3]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%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))))
```

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

[4]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
```

```
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

2 Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

[5]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720745909845e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization loss.

[6]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

[7]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward␣
↪pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
↪verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
↪grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447656e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

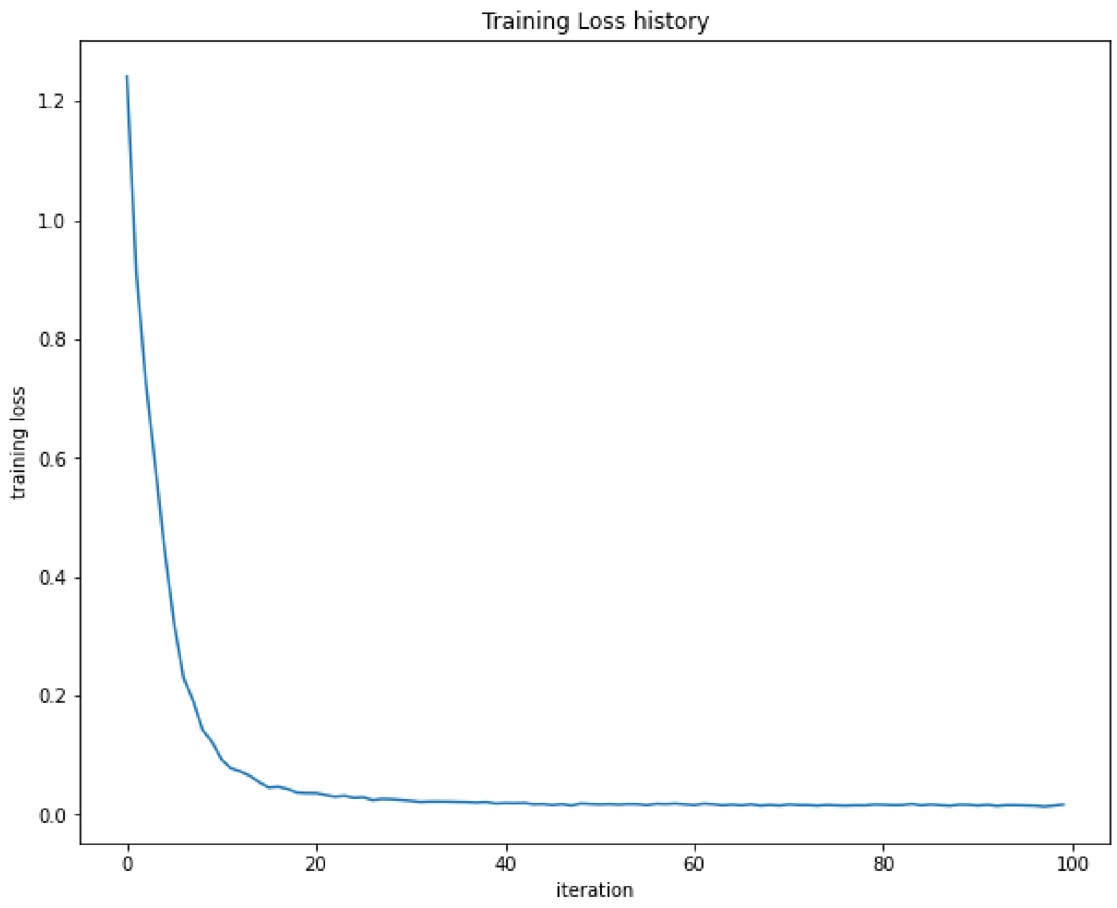
5 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

[9]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.017149607938732048
```



6 Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

[10]:
```
from cs231n.data_utils import load_CIFAR10

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

    # Cleaning up variables to prevent loading data multiple times (which may␣
 ↪cause memory issue)
```

```
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

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

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

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

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

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


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

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
```

```
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

7 Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

[11]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

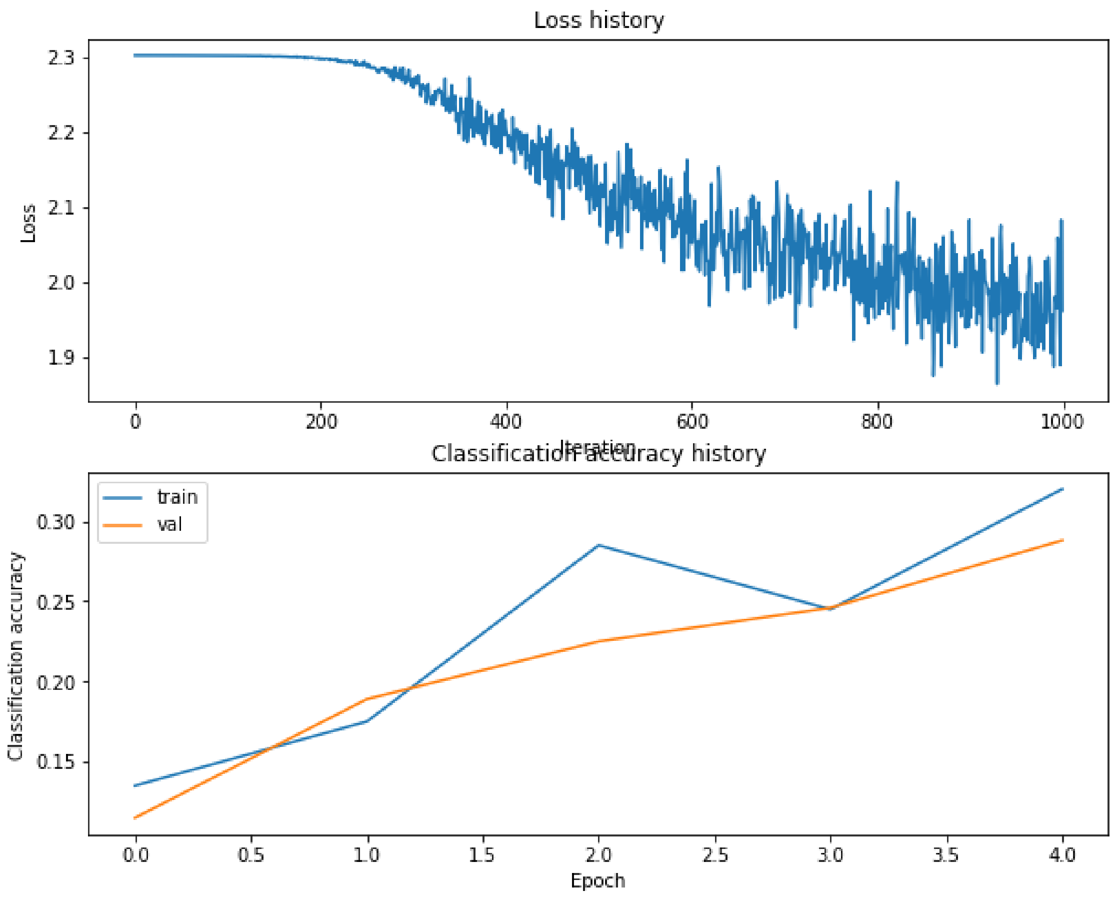
8 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

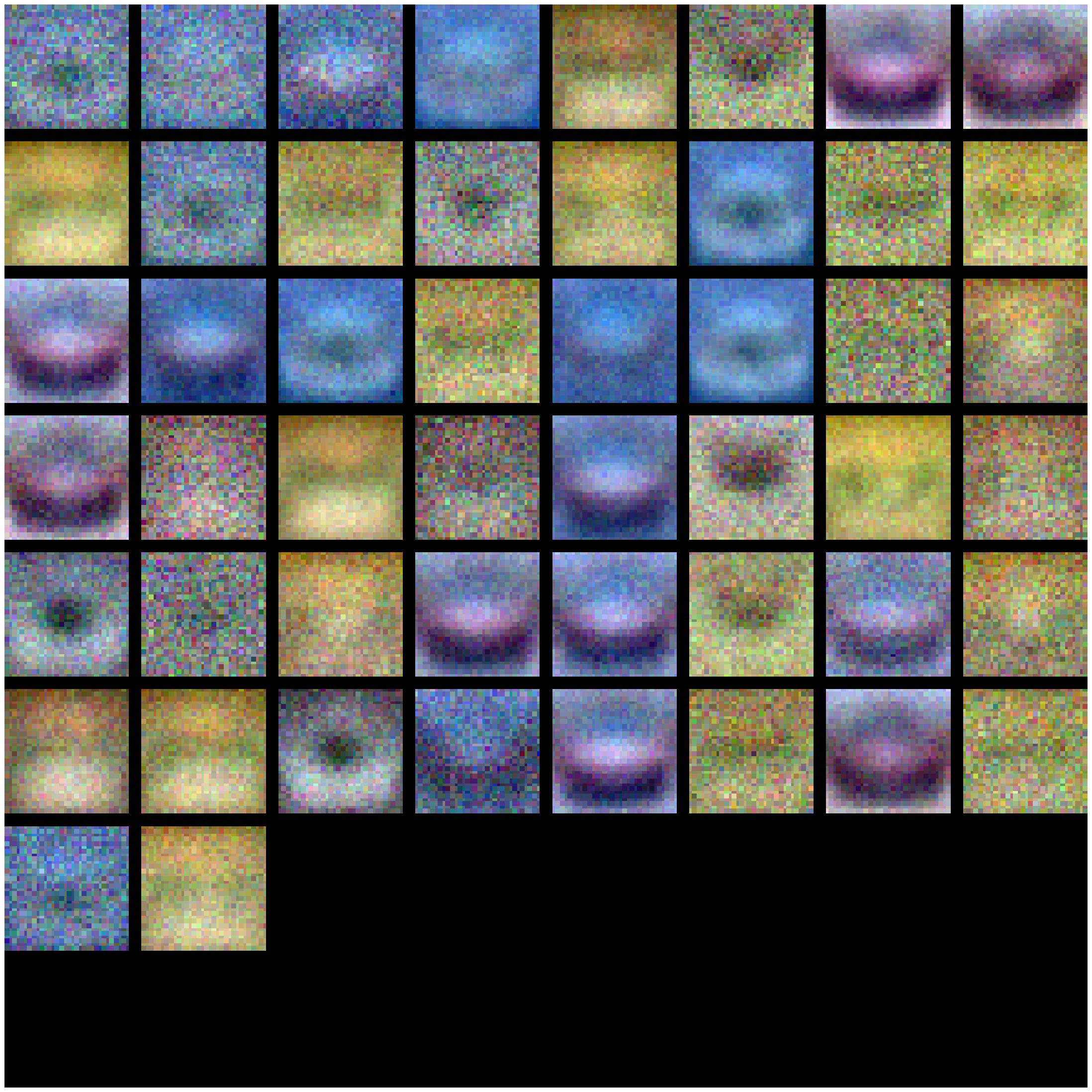
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

[12]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



[13]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

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

show_net_weights(net)
```



9 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, number 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.).

Explain your hyperparameter tuning process below.

YourAnswer : Go through several values for regulation strength, learning rate, number of iterations, hidden layer sizes, and batch sizes.

[18]:
```
best_net = None # store the best model into this

##################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
 ↪#
# model in best_net.                                                           ␣
 ↪#
#                                                                              ␣
 ↪#
# 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 the previous exercises.                         ␣
 ↪#
##################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

best_acc = -1
```

```
best_hs = None
best_lr = None
best_rs = None
regularization_strengths = [0.25, 0.4, 1e-3, 1e-4, 1e-2, 1e-5]
learning_rates = [1e-4, 1e-5, 1e-3, 1e-2, 5e-3, 5e-2]
hidden_sizes = [32, 50, 64, 128, 256, 512]

for lr in learning_rates:
  for rs in regularization_strengths:
    for hs in hidden_sizes:
      net = TwoLayerNet(3072, hs, 10)
      stats = net.train(X_train, y_train, X_val, y_val, learning_rate=lr, reg =␣
 ↪rs, num_iters=1000)
      train_acc = stats['train_acc_history'][-1]
      val_acc = stats['val_acc_history'][-1]
      loss = stats['loss_history'][-1]
      print('hs %d | lr %0.3e | rs %0.3e | loss  %0.3e | train_acc %f | val_acc␣
 ↪%f'%(hs, lr, rs, loss, train_acc, val_acc))
      if val_acc > best_acc:
        best_acc = val_acc
        best_net = net
        best_hs = hs
        best_lr = lr
        best_rs = rs

print('best val_acc = %f for hs %d | lr %e | reg %e'␣
 ↪%(best_acc,best_hs,best_lr,best_rs))

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

```
hs 32 | lr 1.000e-04 | rs 2.500e-01 | loss  1.940e+00 | train_acc 0.290000 |
val_acc 0.277000
hs 50 | lr 1.000e-04 | rs 2.500e-01 | loss  1.946e+00 | train_acc 0.230000 |
val_acc 0.279000
hs 64 | lr 1.000e-04 | rs 2.500e-01 | loss  1.947e+00 | train_acc 0.365000 |
val_acc 0.279000
hs 128 | lr 1.000e-04 | rs 2.500e-01 | loss  1.925e+00 | train_acc 0.280000 |
val_acc 0.283000
hs 256 | lr 1.000e-04 | rs 2.500e-01 | loss  1.966e+00 | train_acc 0.375000 |
val_acc 0.308000
hs 512 | lr 1.000e-04 | rs 2.500e-01 | loss  1.941e+00 | train_acc 0.355000 |
val_acc 0.313000
hs 32 | lr 1.000e-04 | rs 4.000e-01 | loss  1.978e+00 | train_acc 0.245000 |
val_acc 0.271000
hs 50 | lr 1.000e-04 | rs 4.000e-01 | loss  1.993e+00 | train_acc 0.315000 |
val_acc 0.285000
hs 64 | lr 1.000e-04 | rs 4.000e-01 | loss  1.994e+00 | train_acc 0.270000 |
val_acc 0.288000
hs 128 | lr 1.000e-04 | rs 4.000e-01 | loss  1.935e+00 | train_acc 0.225000 |
val_acc 0.291000
hs 256 | lr 1.000e-04 | rs 4.000e-01 | loss  1.896e+00 | train_acc 0.325000 |
val_acc 0.307000
hs 512 | lr 1.000e-04 | rs 4.000e-01 | loss  1.897e+00 | train_acc 0.310000 |
val_acc 0.317000
hs 32 | lr 1.000e-04 | rs 1.000e-03 | loss  1.926e+00 | train_acc 0.250000 |
val_acc 0.282000
hs 50 | lr 1.000e-04 | rs 1.000e-03 | loss  1.937e+00 | train_acc 0.270000 |
val_acc 0.273000
hs 64 | lr 1.000e-04 | rs 1.000e-03 | loss  1.935e+00 | train_acc 0.320000 |
val_acc 0.279000
hs 128 | lr 1.000e-04 | rs 1.000e-03 | loss  1.982e+00 | train_acc 0.315000 |
val_acc 0.285000
hs 256 | lr 1.000e-04 | rs 1.000e-03 | loss  1.947e+00 | train_acc 0.310000 |
val_acc 0.309000
hs 512 | lr 1.000e-04 | rs 1.000e-03 | loss  1.978e+00 | train_acc 0.390000 |
val_acc 0.310000
hs 32 | lr 1.000e-04 | rs 1.000e-04 | loss  1.865e+00 | train_acc 0.275000 |
val_acc 0.276000
hs 50 | lr 1.000e-04 | rs 1.000e-04 | loss  1.928e+00 | train_acc 0.240000 |
val_acc 0.279000
hs 64 | lr 1.000e-04 | rs 1.000e-04 | loss  1.964e+00 | train_acc 0.270000 |
val_acc 0.284000
hs 128 | lr 1.000e-04 | rs 1.000e-04 | loss  1.958e+00 | train_acc 0.310000 |
val_acc 0.293000
hs 256 | lr 1.000e-04 | rs 1.000e-04 | loss  1.908e+00 | train_acc 0.330000 |
val_acc 0.309000
hs 512 | lr 1.000e-04 | rs 1.000e-04 | loss  1.835e+00 | train_acc 0.370000 |
val_acc 0.311000
hs 32 | lr 1.000e-04 | rs 1.000e-02 | loss  1.927e+00 | train_acc 0.280000 |
val_acc 0.280000
hs 50 | lr 1.000e-04 | rs 1.000e-02 | loss  1.962e+00 | train_acc 0.285000 |
val_acc 0.277000
hs 64 | lr 1.000e-04 | rs 1.000e-02 | loss  1.951e+00 | train_acc 0.280000 |
val_acc 0.289000
hs 128 | lr 1.000e-04 | rs 1.000e-02 | loss  1.966e+00 | train_acc 0.280000 |
val_acc 0.294000
hs 256 | lr 1.000e-04 | rs 1.000e-02 | loss  1.898e+00 | train_acc 0.280000 |
val_acc 0.315000
hs 512 | lr 1.000e-04 | rs 1.000e-02 | loss  1.872e+00 | train_acc 0.270000 |
val_acc 0.317000
hs 32 | lr 1.000e-04 | rs 1.000e-05 | loss  1.968e+00 | train_acc 0.270000 |
val_acc 0.279000
hs 50 | lr 1.000e-04 | rs 1.000e-05 | loss  1.898e+00 | train_acc 0.315000 |
val_acc 0.277000
hs 64 | lr 1.000e-04 | rs 1.000e-05 | loss  1.958e+00 | train_acc 0.285000 |
val_acc 0.286000
hs 128 | lr 1.000e-04 | rs 1.000e-05 | loss  1.900e+00 | train_acc 0.315000 |
val_acc 0.290000
hs 256 | lr 1.000e-04 | rs 1.000e-05 | loss  1.924e+00 | train_acc 0.310000 |
val_acc 0.307000
hs 512 | lr 1.000e-04 | rs 1.000e-05 | loss  1.844e+00 | train_acc 0.270000 |
val_acc 0.314000
hs 32 | lr 1.000e-05 | rs 2.500e-01 | loss  2.303e+00 | train_acc 0.140000 |
val_acc 0.197000
hs 50 | lr 1.000e-05 | rs 2.500e-01 | loss  2.302e+00 | train_acc 0.240000 |
val_acc 0.229000
hs 64 | lr 1.000e-05 | rs 2.500e-01 | loss  2.303e+00 | train_acc 0.230000 |
val_acc 0.228000
hs 128 | lr 1.000e-05 | rs 2.500e-01 | loss  2.302e+00 | train_acc 0.195000 |
val_acc 0.243000
hs 256 | lr 1.000e-05 | rs 2.500e-01 | loss  2.303e+00 | train_acc 0.215000 |
val_acc 0.232000
hs 512 | lr 1.000e-05 | rs 2.500e-01 | loss  2.303e+00 | train_acc 0.200000 |
val_acc 0.238000
hs 32 | lr 1.000e-05 | rs 4.000e-01 | loss  2.303e+00 | train_acc 0.220000 |
val_acc 0.210000
hs 50 | lr 1.000e-05 | rs 4.000e-01 | loss  2.303e+00 | train_acc 0.230000 |
val_acc 0.220000
hs 64 | lr 1.000e-05 | rs 4.000e-01 | loss  2.303e+00 | train_acc 0.205000 |
val_acc 0.189000
hs 128 | lr 1.000e-05 | rs 4.000e-01 | loss  2.304e+00 | train_acc 0.205000 |
val_acc 0.200000
hs 256 | lr 1.000e-05 | rs 4.000e-01 | loss  2.304e+00 | train_acc 0.220000 |
val_acc 0.247000
hs 512 | lr 1.000e-05 | rs 4.000e-01 | loss  2.306e+00 | train_acc 0.240000 |
val_acc 0.233000
hs 32 | lr 1.000e-05 | rs 1.000e-03 | loss  2.302e+00 | train_acc 0.235000 |
val_acc 0.240000
hs 50 | lr 1.000e-05 | rs 1.000e-03 | loss  2.302e+00 | train_acc 0.240000 |
val_acc 0.203000
hs 64 | lr 1.000e-05 | rs 1.000e-03 | loss  2.302e+00 | train_acc 0.240000 |
val_acc 0.219000
hs 128 | lr 1.000e-05 | rs 1.000e-03 | loss  2.302e+00 | train_acc 0.190000 |
val_acc 0.220000
hs 256 | lr 1.000e-05 | rs 1.000e-03 | loss  2.301e+00 | train_acc 0.235000 |
val_acc 0.256000
hs 512 | lr 1.000e-05 | rs 1.000e-03 | loss  2.298e+00 | train_acc 0.155000 |
val_acc 0.224000
hs 32 | lr 1.000e-05 | rs 1.000e-04 | loss  2.302e+00 | train_acc 0.155000 |
val_acc 0.185000
hs 50 | lr 1.000e-05 | rs 1.000e-04 | loss  2.302e+00 | train_acc 0.200000 |
val_acc 0.189000
hs 64 | lr 1.000e-05 | rs 1.000e-04 | loss  2.302e+00 | train_acc 0.220000 |
val_acc 0.231000
hs 128 | lr 1.000e-05 | rs 1.000e-04 | loss  2.302e+00 | train_acc 0.215000 |
val_acc 0.242000
hs 256 | lr 1.000e-05 | rs 1.000e-04 | loss  2.300e+00 | train_acc 0.220000 |
val_acc 0.227000
hs 512 | lr 1.000e-05 | rs 1.000e-04 | loss  2.299e+00 | train_acc 0.180000 |
val_acc 0.247000
hs 32 | lr 1.000e-05 | rs 1.000e-02 | loss  2.302e+00 | train_acc 0.170000 |
val_acc 0.206000
hs 50 | lr 1.000e-05 | rs 1.000e-02 | loss  2.302e+00 | train_acc 0.175000 |
val_acc 0.179000
hs 64 | lr 1.000e-05 | rs 1.000e-02 | loss  2.302e+00 | train_acc 0.190000 |
val_acc 0.220000
hs 128 | lr 1.000e-05 | rs 1.000e-02 | loss  2.302e+00 | train_acc 0.200000 |
val_acc 0.220000
hs 256 | lr 1.000e-05 | rs 1.000e-02 | loss  2.301e+00 | train_acc 0.245000 |
val_acc 0.236000
hs 512 | lr 1.000e-05 | rs 1.000e-02 | loss  2.299e+00 | train_acc 0.265000 |
val_acc 0.243000
hs 32 | lr 1.000e-05 | rs 1.000e-05 | loss  2.302e+00 | train_acc 0.185000 |
val_acc 0.195000
hs 50 | lr 1.000e-05 | rs 1.000e-05 | loss  2.302e+00 | train_acc 0.235000 |
val_acc 0.223000
hs 64 | lr 1.000e-05 | rs 1.000e-05 | loss  2.302e+00 | train_acc 0.205000 |
val_acc 0.196000
hs 128 | lr 1.000e-05 | rs 1.000e-05 | loss  2.302e+00 | train_acc 0.185000 |
val_acc 0.232000
hs 256 | lr 1.000e-05 | rs 1.000e-05 | loss  2.301e+00 | train_acc 0.285000 |
val_acc 0.271000
hs 512 | lr 1.000e-05 | rs 1.000e-05 | loss  2.299e+00 | train_acc 0.195000 |
val_acc 0.240000
hs 32 | lr 1.000e-03 | rs 2.500e-01 | loss  1.599e+00 | train_acc 0.560000 |
val_acc 0.444000
hs 50 | lr 1.000e-03 | rs 2.500e-01 | loss  1.579e+00 | train_acc 0.550000 |
val_acc 0.467000
hs 64 | lr 1.000e-03 | rs 2.500e-01 | loss  1.547e+00 | train_acc 0.535000 |
val_acc 0.494000
hs 128 | lr 1.000e-03 | rs 2.500e-01 | loss  1.533e+00 | train_acc 0.540000 |
val_acc 0.465000
hs 256 | lr 1.000e-03 | rs 2.500e-01 | loss  1.372e+00 | train_acc 0.600000 |
val_acc 0.470000
hs 512 | lr 1.000e-03 | rs 2.500e-01 | loss  1.496e+00 | train_acc 0.600000 |
val_acc 0.488000
hs 32 | lr 1.000e-03 | rs 4.000e-01 | loss  1.630e+00 | train_acc 0.530000 |
val_acc 0.454000
hs 50 | lr 1.000e-03 | rs 4.000e-01 | loss  1.633e+00 | train_acc 0.515000 |
val_acc 0.458000
hs 64 | lr 1.000e-03 | rs 4.000e-01 | loss  1.512e+00 | train_acc 0.550000 |
val_acc 0.457000
hs 128 | lr 1.000e-03 | rs 4.000e-01 | loss  1.574e+00 | train_acc 0.580000 |
val_acc 0.471000
hs 256 | lr 1.000e-03 | rs 4.000e-01 | loss  1.562e+00 | train_acc 0.555000 |
val_acc 0.484000
hs 512 | lr 1.000e-03 | rs 4.000e-01 | loss  1.431e+00 | train_acc 0.560000 |
val_acc 0.485000
hs 32 | lr 1.000e-03 | rs 1.000e-03 | loss  1.399e+00 | train_acc 0.505000 |
val_acc 0.455000
hs 50 | lr 1.000e-03 | rs 1.000e-03 | loss  1.459e+00 | train_acc 0.570000 |
val_acc 0.467000
hs 64 | lr 1.000e-03 | rs 1.000e-03 | loss  1.478e+00 | train_acc 0.545000 |
val_acc 0.485000
hs 128 | lr 1.000e-03 | rs 1.000e-03 | loss  1.434e+00 | train_acc 0.570000 |
val_acc 0.489000
hs 256 | lr 1.000e-03 | rs 1.000e-03 | loss  1.584e+00 | train_acc 0.600000 |
val_acc 0.473000
hs 512 | lr 1.000e-03 | rs 1.000e-03 | loss  1.390e+00 | train_acc 0.595000 |
val_acc 0.456000
hs 32 | lr 1.000e-03 | rs 1.000e-04 | loss  1.418e+00 | train_acc 0.540000 |
val_acc 0.463000
hs 50 | lr 1.000e-03 | rs 1.000e-04 | loss  1.380e+00 | train_acc 0.490000 |
val_acc 0.470000
hs 64 | lr 1.000e-03 | rs 1.000e-04 | loss  1.480e+00 | train_acc 0.605000 |
val_acc 0.471000
hs 128 | lr 1.000e-03 | rs 1.000e-04 | loss  1.585e+00 | train_acc 0.605000 |
val_acc 0.473000
hs 256 | lr 1.000e-03 | rs 1.000e-04 | loss  1.412e+00 | train_acc 0.635000 |
val_acc 0.499000
hs 512 | lr 1.000e-03 | rs 1.000e-04 | loss  1.360e+00 | train_acc 0.605000 |
val_acc 0.503000
hs 32 | lr 1.000e-03 | rs 1.000e-02 | loss  1.357e+00 | train_acc 0.570000 |
val_acc 0.447000
hs 50 | lr 1.000e-03 | rs 1.000e-02 | loss  1.476e+00 | train_acc 0.555000 |
val_acc 0.492000
hs 64 | lr 1.000e-03 | rs 1.000e-02 | loss  1.452e+00 | train_acc 0.575000 |
val_acc 0.478000
hs 128 | lr 1.000e-03 | rs 1.000e-02 | loss  1.417e+00 | train_acc 0.585000 |
val_acc 0.495000
hs 256 | lr 1.000e-03 | rs 1.000e-02 | loss  1.554e+00 | train_acc 0.550000 |
val_acc 0.494000
hs 512 | lr 1.000e-03 | rs 1.000e-02 | loss  1.391e+00 | train_acc 0.595000 |
val_acc 0.503000
hs 32 | lr 1.000e-03 | rs 1.000e-05 | loss  1.541e+00 | train_acc 0.560000 |
val_acc 0.462000
hs 50 | lr 1.000e-03 | rs 1.000e-05 | loss  1.459e+00 | train_acc 0.580000 |
val_acc 0.466000
hs 64 | lr 1.000e-03 | rs 1.000e-05 | loss  1.470e+00 | train_acc 0.570000 |
val_acc 0.500000
hs 128 | lr 1.000e-03 | rs 1.000e-05 | loss  1.347e+00 | train_acc 0.590000 |
val_acc 0.464000
hs 256 | lr 1.000e-03 | rs 1.000e-05 | loss  1.403e+00 | train_acc 0.560000 |
val_acc 0.478000
hs 512 | lr 1.000e-03 | rs 1.000e-05 | loss  1.270e+00 | train_acc 0.595000 |
val_acc 0.497000

/content/cs231n/classifiers/neural_net.py:106: RuntimeWarning: divide by zero
encountered in log
  log = -np.log(value)
/content/cs231n/classifiers/neural_net.py:103: RuntimeWarning: overflow
encountered in exp
  total = np.sum(np.exp(scores), axis = 1)
/content/cs231n/classifiers/neural_net.py:104: RuntimeWarning: overflow
encountered in exp
  num = np.exp(scores[range(N),y])
/content/cs231n/classifiers/neural_net.py:105: RuntimeWarning: invalid value
encountered in true_divide
  value = num / total
/content/cs231n/classifiers/neural_net.py:122: RuntimeWarning: overflow
encountered in exp
  temp = (np.exp(scores)/total.reshape(-1,1))
/content/cs231n/classifiers/neural_net.py:122: RuntimeWarning: invalid value
encountered in true_divide
  temp = (np.exp(scores)/total.reshape(-1,1))
/content/cs231n/classifiers/neural_net.py:127: RuntimeWarning: invalid value
encountered in less_equal
  dh1[H1<=0] = 0

hs 32 | lr 1.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 50 | lr 1.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.080000 | val_acc
0.087000
hs 64 | lr 1.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.085000 | val_acc
0.087000
hs 128 | lr 1.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.080000 | val_acc
0.087000
hs 256 | lr 1.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 512 | lr 1.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.130000 | val_acc
0.087000
hs 32 | lr 1.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.070000 | val_acc
0.087000
hs 50 | lr 1.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 64 | lr 1.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.115000 | val_acc
0.087000
```

```
hs 128 | lr 1.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.085000 | val_acc
0.087000
hs 256 | lr 1.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 512 | lr 1.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 32 | lr 1.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 50 | lr 1.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 64 | lr 1.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 128 | lr 1.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 256 | lr 1.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.070000 | val_acc
0.087000
hs 512 | lr 1.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.130000 | val_acc
0.087000
hs 32 | lr 1.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 50 | lr 1.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 64 | lr 1.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 128 | lr 1.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 256 | lr 1.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.055000 | val_acc
0.087000
hs 512 | lr 1.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 32 | lr 1.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 50 | lr 1.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 64 | lr 1.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.140000 | val_acc
0.087000
hs 128 | lr 1.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 256 | lr 1.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.090000 | val_acc
0.087000
hs 512 | lr 1.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 32 | lr 1.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 50 | lr 1.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.135000 | val_acc
0.087000
hs 64 | lr 1.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.100000 | val_acc
0.087000
```

```
hs 128 | lr 1.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 256 | lr 1.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 512 | lr 1.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 32 | lr 5.000e-03 | rs 2.500e-01 | loss  nan | train_acc 0.125000 | val_acc
0.087000
hs 50 | lr 5.000e-03 | rs 2.500e-01 | loss  nan | train_acc 0.085000 | val_acc
0.087000
hs 64 | lr 5.000e-03 | rs 2.500e-01 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 128 | lr 5.000e-03 | rs 2.500e-01 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 256 | lr 5.000e-03 | rs 2.500e-01 | loss  nan | train_acc 0.120000 | val_acc
0.087000
hs 512 | lr 5.000e-03 | rs 2.500e-01 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 32 | lr 5.000e-03 | rs 4.000e-01 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 50 | lr 5.000e-03 | rs 4.000e-01 | loss  nan | train_acc 0.140000 | val_acc
0.087000
hs 64 | lr 5.000e-03 | rs 4.000e-01 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 128 | lr 5.000e-03 | rs 4.000e-01 | loss  nan | train_acc 0.090000 | val_acc
0.087000
hs 256 | lr 5.000e-03 | rs 4.000e-01 | loss  nan | train_acc 0.145000 | val_acc
0.087000
hs 512 | lr 5.000e-03 | rs 4.000e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 32 | lr 5.000e-03 | rs 1.000e-03 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 50 | lr 5.000e-03 | rs 1.000e-03 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 64 | lr 5.000e-03 | rs 1.000e-03 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 128 | lr 5.000e-03 | rs 1.000e-03 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 256 | lr 5.000e-03 | rs 1.000e-03 | loss  nan | train_acc 0.125000 | val_acc
0.087000
hs 512 | lr 5.000e-03 | rs 1.000e-03 | loss  nan | train_acc 0.130000 | val_acc
0.087000
hs 32 | lr 5.000e-03 | rs 1.000e-04 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 50 | lr 5.000e-03 | rs 1.000e-04 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 64 | lr 5.000e-03 | rs 1.000e-04 | loss  nan | train_acc 0.130000 | val_acc
0.087000
```

```
hs 128 | lr 5.000e-03 | rs 1.000e-04 | loss  nan | train_acc 0.125000 | val_acc
0.087000
hs 256 | lr 5.000e-03 | rs 1.000e-04 | loss  nan | train_acc 0.120000 | val_acc
0.087000
hs 512 | lr 5.000e-03 | rs 1.000e-04 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 32 | lr 5.000e-03 | rs 1.000e-02 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 50 | lr 5.000e-03 | rs 1.000e-02 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 64 | lr 5.000e-03 | rs 1.000e-02 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 128 | lr 5.000e-03 | rs 1.000e-02 | loss  nan | train_acc 0.060000 | val_acc
0.087000
hs 256 | lr 5.000e-03 | rs 1.000e-02 | loss  nan | train_acc 0.135000 | val_acc
0.087000
hs 512 | lr 5.000e-03 | rs 1.000e-02 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 32 | lr 5.000e-03 | rs 1.000e-05 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 50 | lr 5.000e-03 | rs 1.000e-05 | loss  nan | train_acc 0.125000 | val_acc
0.087000
hs 64 | lr 5.000e-03 | rs 1.000e-05 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 128 | lr 5.000e-03 | rs 1.000e-05 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 256 | lr 5.000e-03 | rs 1.000e-05 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 512 | lr 5.000e-03 | rs 1.000e-05 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 32 | lr 5.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 50 | lr 5.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.055000 | val_acc
0.087000
hs 64 | lr 5.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 128 | lr 5.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.135000 | val_acc
0.087000
hs 256 | lr 5.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 512 | lr 5.000e-02 | rs 2.500e-01 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 32 | lr 5.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 50 | lr 5.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.065000 | val_acc
0.087000
hs 64 | lr 5.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.110000 | val_acc
0.087000
```

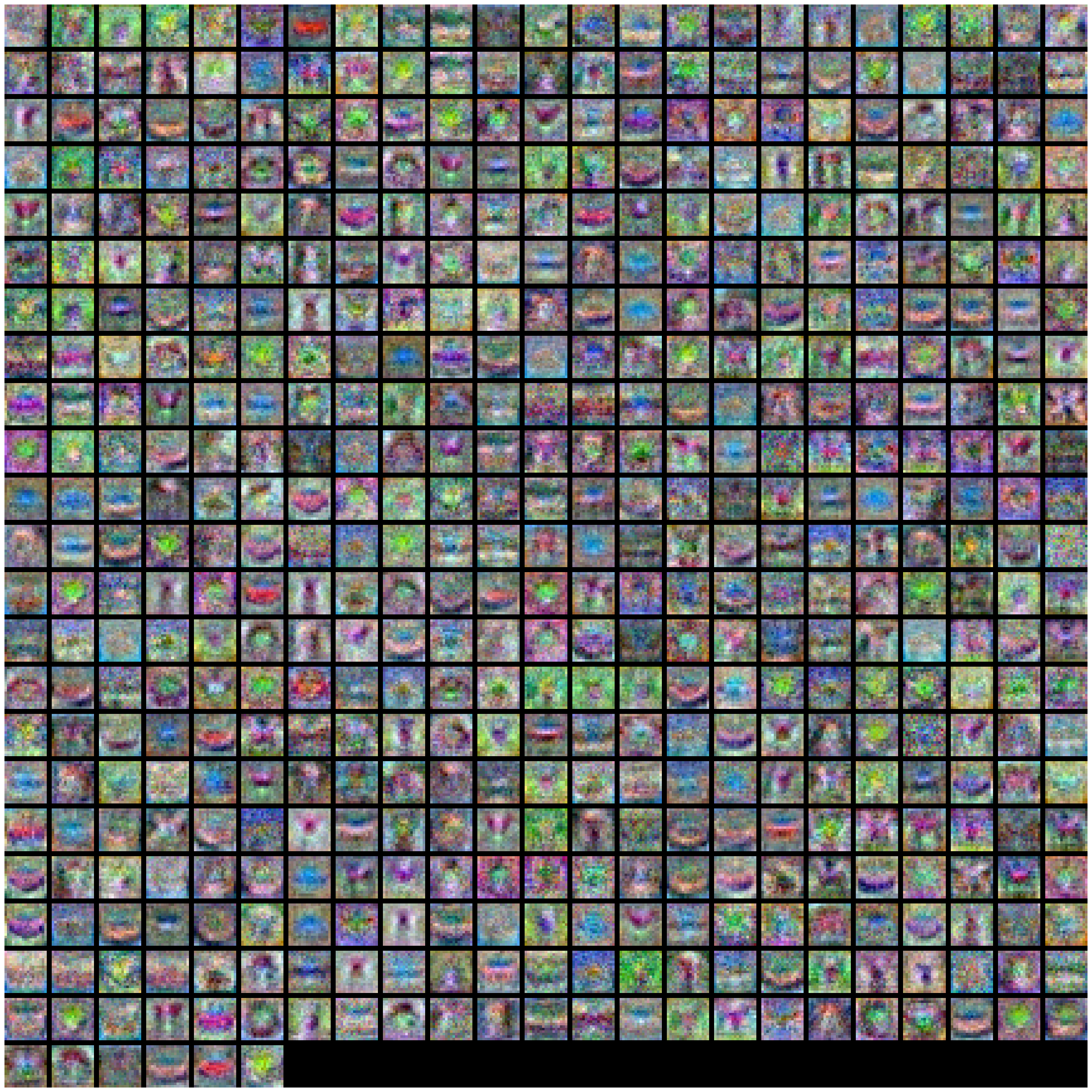
```
hs 128 | lr 5.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 256 | lr 5.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 512 | lr 5.000e-02 | rs 4.000e-01 | loss  nan | train_acc 0.105000 | val_acc
0.087000
hs 32 | lr 5.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.150000 | val_acc
0.087000
hs 50 | lr 5.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.100000 | val_acc
0.087000
hs 64 | lr 5.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.130000 | val_acc
0.087000
hs 128 | lr 5.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 256 | lr 5.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.085000 | val_acc
0.087000
hs 512 | lr 5.000e-02 | rs 1.000e-03 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 32 | lr 5.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.120000 | val_acc
0.087000
hs 50 | lr 5.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.125000 | val_acc
0.087000
hs 64 | lr 5.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 128 | lr 5.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.070000 | val_acc
0.087000
hs 256 | lr 5.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.130000 | val_acc
0.087000
hs 512 | lr 5.000e-02 | rs 1.000e-04 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 32 | lr 5.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.090000 | val_acc
0.087000
hs 50 | lr 5.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.075000 | val_acc
0.087000
hs 64 | lr 5.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.090000 | val_acc
0.087000
hs 128 | lr 5.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.090000 | val_acc
0.087000
hs 256 | lr 5.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 512 | lr 5.000e-02 | rs 1.000e-02 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 32 | lr 5.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.095000 | val_acc
0.087000
hs 50 | lr 5.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.080000 | val_acc
0.087000
hs 64 | lr 5.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.115000 | val_acc
0.087000
```

```
hs 128 | lr 5.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.085000 | val_acc
0.087000
hs 256 | lr 5.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.110000 | val_acc
0.087000
hs 512 | lr 5.000e-02 | rs 1.000e-05 | loss  nan | train_acc 0.105000 | val_acc
0.087000
best val_acc = 0.503000 for hs 512 | lr 1.000000e-03 | reg 1.000000e-04
```

[19]:
```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.526
```

[20]:
```
# Visualize the weights of the best network
show_net_weights(best_net)
```



10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

[21]:
```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.481
```

Inline Question

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

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

$YourAnswer:$ 1, 2

$YourExplanation:$ More data means a more ifnely tuned network, incresing accuracy. More hidden units means more parameters and thus tuning of the network. Increasing regulation strength doesn't necessarily increase accuracy.

---

11 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[8]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/neural_net.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```

features

September 12, 2020

[17]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'cs231n/assignment1/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-09-12 21:10:11--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  30.1MB/s    in 6.0s

2020-09-12 21:10:17 (27.0 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

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.

[18]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


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

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

```
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.

[19]:
```
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.

[20]:
```
from cs231n.features import *

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

```
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

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

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

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

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

[21]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3]
regularization_strengths = [5e4, 5e5, 5e6, 0.2]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        ␣
 ↪#
```

```
# Use the validation set to set the learning rate and regularization strength.␣
 ↪#
# This should be identical to the validation that you did for the SVM; save    ␣
 ↪#
# the best trained classifer in best_svm. You might also want to play          ␣
 ↪#
# with different numbers of bins in the color histogram. If you are careful    ␣
 ↪#
# you should be able to get accuracy of near 0.44 on the validation set.       ␣
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
  for rs in regularization_strengths:
    svm = LinearSVM()
    loss_hist = svm.train(X_train_feats, y_train, learning_rate = lr, reg = rs,␣
 ↪num_iters = 1000, verbose = True)
    train_acc = np.mean(svm.predict(X_train_feats) == y_train)
    val_acc = np.mean(svm.predict(X_val_feats) == y_val)
    results[(lr, rs)] = (train_acc, val_acc)
    if val_acc > best_val:
      best_val = val_acc
      best_svm = svm

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

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

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

```
iteration 0 / 1000: loss 88.353818
iteration 100 / 1000: loss 86.787241
iteration 200 / 1000: loss 85.220133
iteration 300 / 1000: loss 83.733979
iteration 400 / 1000: loss 82.247815
iteration 500 / 1000: loss 80.796982
iteration 600 / 1000: loss 79.377804
iteration 700 / 1000: loss 77.978453
iteration 800 / 1000: loss 76.616845
iteration 900 / 1000: loss 75.279044
iteration 0 / 1000: loss 806.852615
iteration 100 / 1000: loss 662.162996
iteration 200 / 1000: loss 543.718801
iteration 300 / 1000: loss 446.736300
iteration 400 / 1000: loss 367.357088
iteration 500 / 1000: loss 302.366277
iteration 600 / 1000: loss 249.161539
iteration 700 / 1000: loss 205.607407
iteration 800 / 1000: loss 169.946125
iteration 900 / 1000: loss 140.771190
iteration 0 / 1000: loss 7797.846499
iteration 100 / 1000: loss 1052.538908
iteration 200 / 1000: loss 148.815792
iteration 300 / 1000: loss 27.732531
iteration 400 / 1000: loss 11.509808
iteration 500 / 1000: loss 9.336264
iteration 600 / 1000: loss 9.045032
iteration 700 / 1000: loss 9.006039
iteration 800 / 1000: loss 9.000806
iteration 900 / 1000: loss 9.000105
iteration 0 / 1000: loss 8.982259
iteration 100 / 1000: loss 9.007798
iteration 200 / 1000: loss 8.985338
iteration 300 / 1000: loss 8.987720
iteration 400 / 1000: loss 9.018746
iteration 500 / 1000: loss 8.990703
iteration 600 / 1000: loss 9.011541
iteration 700 / 1000: loss 8.987880
iteration 800 / 1000: loss 8.998243
iteration 900 / 1000: loss 9.008185
iteration 0 / 1000: loss 86.916390
iteration 100 / 1000: loss 72.779240
iteration 200 / 1000: loss 61.212383
iteration 300 / 1000: loss 51.736561
iteration 400 / 1000: loss 43.993528
iteration 500 / 1000: loss 37.657554
iteration 600 / 1000: loss 32.453648
iteration 700 / 1000: loss 28.188579
iteration 800 / 1000: loss 24.715339
iteration 900 / 1000: loss 21.865945
iteration 0 / 1000: loss 803.142733
iteration 100 / 1000: loss 115.400133
iteration 200 / 1000: loss 23.255258
iteration 300 / 1000: loss 10.909792
iteration 400 / 1000: loss 9.255921
iteration 500 / 1000: loss 9.034237
iteration 600 / 1000: loss 9.004551
iteration 700 / 1000: loss 9.000569
iteration 800 / 1000: loss 9.000045
iteration 900 / 1000: loss 8.999978
iteration 0 / 1000: loss 7888.655801
iteration 100 / 1000: loss 9.000003
iteration 200 / 1000: loss 8.999997
iteration 300 / 1000: loss 8.999996
iteration 400 / 1000: loss 8.999995
iteration 500 / 1000: loss 8.999997
iteration 600 / 1000: loss 8.999997
iteration 700 / 1000: loss 8.999997
iteration 800 / 1000: loss 8.999996
iteration 900 / 1000: loss 8.999996
iteration 0 / 1000: loss 9.012201
iteration 100 / 1000: loss 9.014886
iteration 200 / 1000: loss 9.005605
iteration 300 / 1000: loss 8.977696
iteration 400 / 1000: loss 9.003480
iteration 500 / 1000: loss 9.002560
iteration 600 / 1000: loss 9.011407
iteration 700 / 1000: loss 8.998746
iteration 800 / 1000: loss 9.001855
iteration 900 / 1000: loss 9.009636
iteration 0 / 1000: loss 88.274264
iteration 100 / 1000: loss 19.617706
iteration 200 / 1000: loss 10.421427
iteration 300 / 1000: loss 9.190278
iteration 400 / 1000: loss 9.025283
iteration 500 / 1000: loss 9.003070
iteration 600 / 1000: loss 8.999999
iteration 700 / 1000: loss 8.999709
iteration 800 / 1000: loss 8.999691
iteration 900 / 1000: loss 8.999686
iteration 0 / 1000: loss 812.260342
iteration 100 / 1000: loss 8.999968
iteration 200 / 1000: loss 8.999961
iteration 300 / 1000: loss 8.999961
iteration 400 / 1000: loss 8.999966
iteration 500 / 1000: loss 8.999972
iteration 600 / 1000: loss 8.999975
iteration 700 / 1000: loss 8.999965
iteration 800 / 1000: loss 8.999960
iteration 900 / 1000: loss 8.999969
iteration 0 / 1000: loss 7820.884383
iteration 100 / 1000: loss 9.000000
iteration 200 / 1000: loss 9.000001
iteration 300 / 1000: loss 9.000000
iteration 400 / 1000: loss 9.000001
iteration 500 / 1000: loss 9.000001
iteration 600 / 1000: loss 9.000000
iteration 700 / 1000: loss 9.000002
iteration 800 / 1000: loss 9.000000
iteration 900 / 1000: loss 9.000000
iteration 0 / 1000: loss 9.012997
iteration 100 / 1000: loss 8.998785
iteration 200 / 1000: loss 9.011950
iteration 300 / 1000: loss 8.999290
iteration 400 / 1000: loss 8.992678
iteration 500 / 1000: loss 8.989622
iteration 600 / 1000: loss 8.994330
iteration 700 / 1000: loss 8.994135
iteration 800 / 1000: loss 8.989599
iteration 900 / 1000: loss 8.987054
iteration 0 / 1000: loss 83.008978
iteration 100 / 1000: loss 8.999543
iteration 200 / 1000: loss 8.999722
iteration 300 / 1000: loss 8.999780
iteration 400 / 1000: loss 8.999608
iteration 500 / 1000: loss 8.999698
iteration 600 / 1000: loss 8.999592
iteration 700 / 1000: loss 8.999797
iteration 800 / 1000: loss 8.999721
iteration 900 / 1000: loss 8.999709
iteration 0 / 1000: loss 737.804469
iteration 100 / 1000: loss 9.000011
iteration 200 / 1000: loss 9.000021
iteration 300 / 1000: loss 9.000019
iteration 400 / 1000: loss 9.000011
iteration 500 / 1000: loss 9.000018
iteration 600 / 1000: loss 9.000015
iteration 700 / 1000: loss 8.999997
iteration 800 / 1000: loss 9.000017
iteration 900 / 1000: loss 9.000001
iteration 0 / 1000: loss 7643.514001
iteration 100 / 1000: loss 538620873280810287609190486479171911410790041021808022
3222388987984352545039602958794046533584553897155636691814256457762992796661065
86245295352208968104986981278215157234168719760002566663123763 2.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan

/content/cs231n/classifiers/linear_svm.py:92: RuntimeWarning: overflow
encountered in double_scalars
  loss += reg*np.sum(W**2)
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```

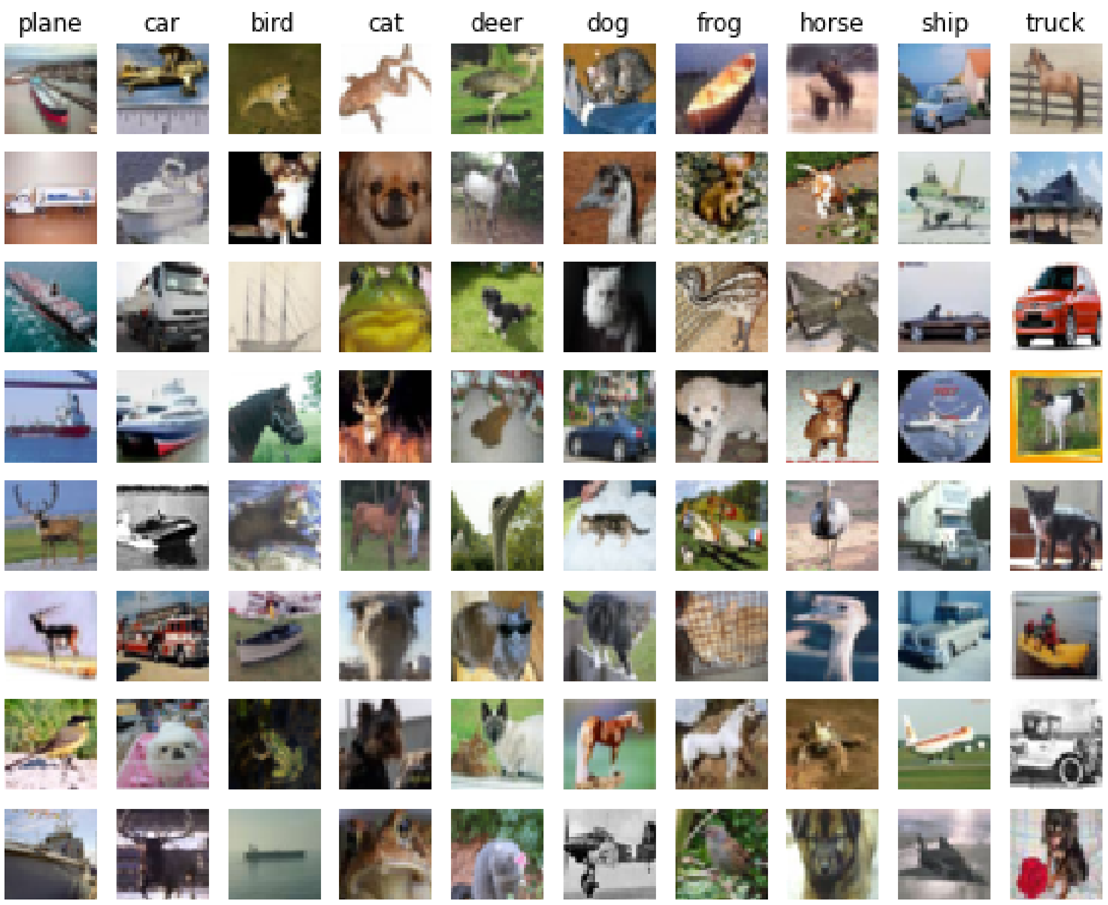
```
/content/cs231n/classifiers/linear_svm.py:92: RuntimeWarning: overflow
encountered in square
  loss += reg*np.sum(W**2)
/content/cs231n/classifiers/linear_svm.py:113: RuntimeWarning: overflow
encountered in multiply
  dW += 2*reg*W
/content/cs231n/classifiers/linear_svm.py:87: RuntimeWarning: invalid value
encountered in matmul
  scores = X @ W
/content/cs231n/classifiers/linear_svm.py:89: RuntimeWarning: invalid value
encountered in less_equal
  margin[margin<=0] = 0
/content/cs231n/classifiers/linear_svm.py:108: RuntimeWarning: invalid value
encountered in greater
  margin[margin>0] = 1

iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 8.988153
iteration 100 / 1000: loss 8.985070
iteration 200 / 1000: loss 8.985438
iteration 300 / 1000: loss 8.970693
iteration 400 / 1000: loss 8.953956
iteration 500 / 1000: loss 8.954373
iteration 600 / 1000: loss 8.972477
iteration 700 / 1000: loss 8.959694
iteration 800 / 1000: loss 8.941550
iteration 900 / 1000: loss 8.909604
iteration 0 / 1000: loss 84.414757
iteration 100 / 1000: loss 9.000078
iteration 200 / 1000: loss 9.000232
iteration 300 / 1000: loss 9.000026
iteration 400 / 1000: loss 9.000061
iteration 500 / 1000: loss 9.000175
iteration 600 / 1000: loss 9.000095
iteration 700 / 1000: loss 9.000133
iteration 800 / 1000: loss 9.000008
iteration 900 / 1000: loss 9.000116
iteration 0 / 1000: loss 786.934764
iteration 100 / 1000: loss 54883508377202913396808907830545025942218098151696549082401158806017289901388000191476725138212625944482297982147083224626893556124162897612689800880767694211890865722799215326090078747713601536.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 7931.047192
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss nan
iteration 300 / 1000: loss nan
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 9.010539
iteration 100 / 1000: loss 8.923147
iteration 200 / 1000: loss 8.868628
iteration 300 / 1000: loss 8.789251
iteration 400 / 1000: loss 8.730937
iteration 500 / 1000: loss 8.610338
iteration 600 / 1000: loss 8.533673
iteration 700 / 1000: loss 8.465239
iteration 800 / 1000: loss 8.408661
iteration 900 / 1000: loss 8.318086
iteration 0 / 1000: loss 88.596252
iteration 100 / 1000: loss 56157127640998356825415104335903681980338121424518825
91285302558831696375448821196379884430130450284950159252812009446953400441205636
831718225719554055140597072208812100524074233991017279782912.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 798.899801
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss nan
iteration 300 / 1000: loss nan
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 7973.409077
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss nan
iteration 300 / 1000: loss nan
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 9.016746
iteration 100 / 1000: loss 8.308946
iteration 200 / 1000: loss 7.169303
iteration 300 / 1000: loss 6.887974
iteration 400 / 1000: loss 6.022709
iteration 500 / 1000: loss 5.611827
iteration 600 / 1000: loss 5.315515
iteration 700 / 1000: loss 4.995259
iteration 800 / 1000: loss 4.843462
iteration 900 / 1000: loss 4.929774
iteration 0 / 1000: loss 85.259354
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss nan
iteration 300 / 1000: loss nan
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 802.564425
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss nan
iteration 300 / 1000: loss nan
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 7826.718521
iteration 100 / 1000: loss nan
iteration 200 / 1000: loss nan

/content/cs231n/classifiers/linear_svm.py:88: RuntimeWarning: invalid value
encountered in subtract
  margin = scores - scores[np.arange(num_train), y].reshape(-1,1) + 1

iteration 300 / 1000: loss nan
iteration 400 / 1000: loss nan
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
iteration 0 / 1000: loss 8.999663
iteration 100 / 1000: loss 4.721098
iteration 200 / 1000: loss 4.571261
iteration 300 / 1000: loss 4.300628
iteration 400 / 1000: loss 3.707152
iteration 500 / 1000: loss 4.374091
iteration 600 / 1000: loss 3.601590
iteration 700 / 1000: loss 3.892744
iteration 800 / 1000: loss 3.321862
iteration 900 / 1000: loss 4.016249
lr 1.000000e-09 reg 2.000000e-01 train accuracy: 0.111959 val accuracy: 0.120000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.095959 val accuracy: 0.090000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.129816 val accuracy: 0.141000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.302041 val accuracy: 0.275000
lr 1.000000e-08 reg 2.000000e-01 train accuracy: 0.091755 val accuracy: 0.090000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.093510 val accuracy: 0.101000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416980 val accuracy: 0.414000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.410612 val accuracy: 0.404000
lr 1.000000e-07 reg 2.000000e-01 train accuracy: 0.105143 val accuracy: 0.095000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.413102 val accuracy: 0.418000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.413265 val accuracy: 0.429000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.329265 val accuracy: 0.360000
lr 1.000000e-06 reg 2.000000e-01 train accuracy: 0.238102 val accuracy: 0.264000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.412490 val accuracy: 0.401000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.318755 val accuracy: 0.318000
lr 1.000000e-06 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-05 reg 2.000000e-01 train accuracy: 0.400755 val accuracy: 0.411000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.344082 val accuracy: 0.350000
lr 1.000000e-05 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-05 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 2.000000e-01 train accuracy: 0.439714 val accuracy: 0.442000
lr 1.000000e-04 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 2.000000e-01 train accuracy: 0.489469 val accuracy: 0.478000
lr 1.000000e-03 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.478000
```

[22]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
↪0.40
```

```
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.476
```

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

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



1.3.1 Inline question 1:

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

Your Answer : Some of them, such as classifyin trucks as cars make sense. Also, many of the misclassified 'planes' had lots of blue which is common as planes are in the air. Several of the misclassifications don't make sense, for example, the horse classified as a truck.

1.4 Neural Network on image features

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

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

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

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

print(X_train_feats.shape)
```

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

[28]:
```
from cs231n.classifiers.neural_net import TwoLayerNet

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

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

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to   ␣
↪#
# cross-validate various parameters as in previous sections. Store your best   ␣
↪#
# model in the best_net variable.                                               ␣
↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

best_acc = -1
best_res = {}
learning_rates = [1e-4, 1e-3, 1e-2, 5e-3, 5e-2, 1e-1, 5e-1]
regularization_strengths = [5e-4, 5e-6, 1e-5, 1e-4, 1e-3, 5e-5]
results = {}
for lr in learning_rates:
  for rs in regularization_strengths:
    res = net.train(X_train_feats, y_train, X_val_feats, y_val, learning_rate =␣
↪lr, reg = rs, num_iters = 1000, verbose=False)
    train_acc = np.mean(net.predict(X_train_feats) == y_train)
    val_acc = np.mean(net.predict(X_val_feats) == y_val)
    results[(lr,rs)] = (train_acc, val_acc)
    print(val_acc)
    if val_acc > best_acc:
      best_acc = val_acc
      best_net = net
      best_res = res

# Print out results.
```

```python
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_acc)

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

```
0.079
0.079
0.079
0.078
0.087
0.087
0.087
0.078
0.078
0.078
0.079
0.079
0.176
0.288
0.377
0.421
0.462
0.481
0.485
0.499
0.507
0.506
0.508
0.511
0.515
0.548
0.564
0.581
0.582
0.592
0.588
0.599
0.596
0.578
0.594
0.579
0.586
0.566
0.564
```

```
0.583
0.589
0.568
lr 1.000000e-04 reg 5.000000e-06 train accuracy: 0.100429 val accuracy: 0.079000
lr 1.000000e-04 reg 1.000000e-05 train accuracy: 0.100429 val accuracy: 0.079000
lr 1.000000e-04 reg 5.000000e-05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 1.000000e-04 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-04 reg 5.000000e-04 train accuracy: 0.100429 val accuracy: 0.079000
lr 1.000000e-04 reg 1.000000e-03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 5.000000e-06 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-03 reg 1.000000e-05 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-03 reg 5.000000e-05 train accuracy: 0.100429 val accuracy: 0.079000
lr 1.000000e-03 reg 1.000000e-04 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-03 reg 5.000000e-04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 1.000000e-03 train accuracy: 0.100429 val accuracy: 0.079000
lr 5.000000e-03 reg 5.000000e-06 train accuracy: 0.503837 val accuracy: 0.499000
lr 5.000000e-03 reg 1.000000e-05 train accuracy: 0.510469 val accuracy: 0.507000
lr 5.000000e-03 reg 5.000000e-05 train accuracy: 0.524571 val accuracy: 0.511000
lr 5.000000e-03 reg 1.000000e-04 train accuracy: 0.515653 val accuracy: 0.506000
lr 5.000000e-03 reg 5.000000e-04 train accuracy: 0.497857 val accuracy: 0.485000
lr 5.000000e-03 reg 1.000000e-03 train accuracy: 0.520286 val accuracy: 0.508000
lr 1.000000e-02 reg 5.000000e-06 train accuracy: 0.269959 val accuracy: 0.288000
lr 1.000000e-02 reg 1.000000e-05 train accuracy: 0.379776 val accuracy: 0.377000
lr 1.000000e-02 reg 5.000000e-05 train accuracy: 0.490776 val accuracy: 0.481000
lr 1.000000e-02 reg 1.000000e-04 train accuracy: 0.441796 val accuracy: 0.421000
lr 1.000000e-02 reg 5.000000e-04 train accuracy: 0.180000 val accuracy: 0.176000
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.473837 val accuracy: 0.462000
lr 5.000000e-02 reg 5.000000e-06 train accuracy: 0.564000 val accuracy: 0.548000
lr 5.000000e-02 reg 1.000000e-05 train accuracy: 0.586959 val accuracy: 0.564000
lr 5.000000e-02 reg 5.000000e-05 train accuracy: 0.645816 val accuracy: 0.592000
lr 5.000000e-02 reg 1.000000e-04 train accuracy: 0.608388 val accuracy: 0.581000
lr 5.000000e-02 reg 5.000000e-04 train accuracy: 0.543531 val accuracy: 0.515000
lr 5.000000e-02 reg 1.000000e-03 train accuracy: 0.625224 val accuracy: 0.582000
lr 1.000000e-01 reg 5.000000e-06 train accuracy: 0.705265 val accuracy: 0.599000
lr 1.000000e-01 reg 1.000000e-05 train accuracy: 0.732061 val accuracy: 0.596000
lr 1.000000e-01 reg 5.000000e-05 train accuracy: 0.791265 val accuracy: 0.579000
lr 1.000000e-01 reg 1.000000e-04 train accuracy: 0.759204 val accuracy: 0.578000
lr 1.000000e-01 reg 5.000000e-04 train accuracy: 0.671694 val accuracy: 0.588000
lr 1.000000e-01 reg 1.000000e-03 train accuracy: 0.769000 val accuracy: 0.594000
lr 5.000000e-01 reg 5.000000e-06 train accuracy: 0.791224 val accuracy: 0.566000
lr 5.000000e-01 reg 1.000000e-05 train accuracy: 0.855000 val accuracy: 0.564000
lr 5.000000e-01 reg 5.000000e-05 train accuracy: 0.869265 val accuracy: 0.568000
lr 5.000000e-01 reg 1.000000e-04 train accuracy: 0.890327 val accuracy: 0.583000
lr 5.000000e-01 reg 5.000000e-04 train accuracy: 0.749224 val accuracy: 0.586000
lr 5.000000e-01 reg 1.000000e-03 train accuracy: 0.826306 val accuracy: 0.589000
0.599
```

[29]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.557
```

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[ ]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = []

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```