knn

September 30, 2024

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

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

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

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

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

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

[3]:
```python
# 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
```

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

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

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

# As a sanity check, we print out the size of the training and test data.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
```

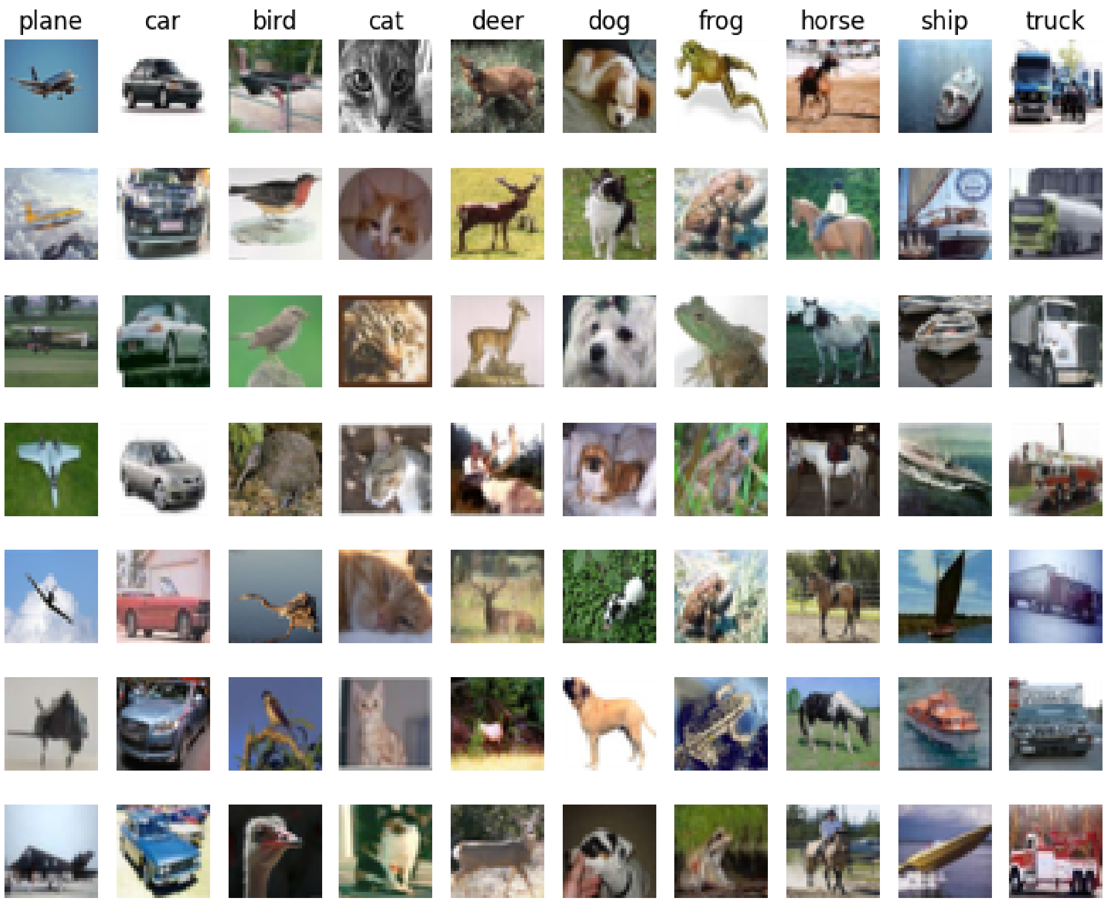
```
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
# PLEASE DO NOT MODIFY THE MARKERS
print('```````````````````````````````````````````````')
```

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

[5]:

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
# PLEASE DO NOT MODIFY THE MARKERS
print('|||||||||||||||||||||||||||||||||||||||||||||||')
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
print('```````````````````````````````````````````````')
```

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



[6]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
print(X_train.shape, X_test.shape)
print('``````````````````````````````````````````````')
```

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

[7]:
```python
from cs231n.classifiers import KNearestNeighbor
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
print('``````````````````````````````````````````````')
```

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

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

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

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

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

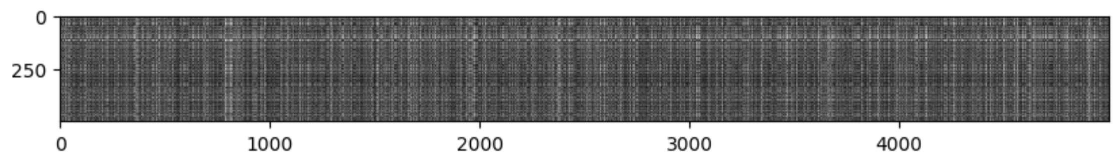
First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

[8]:
```python
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')

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

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

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

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

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




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

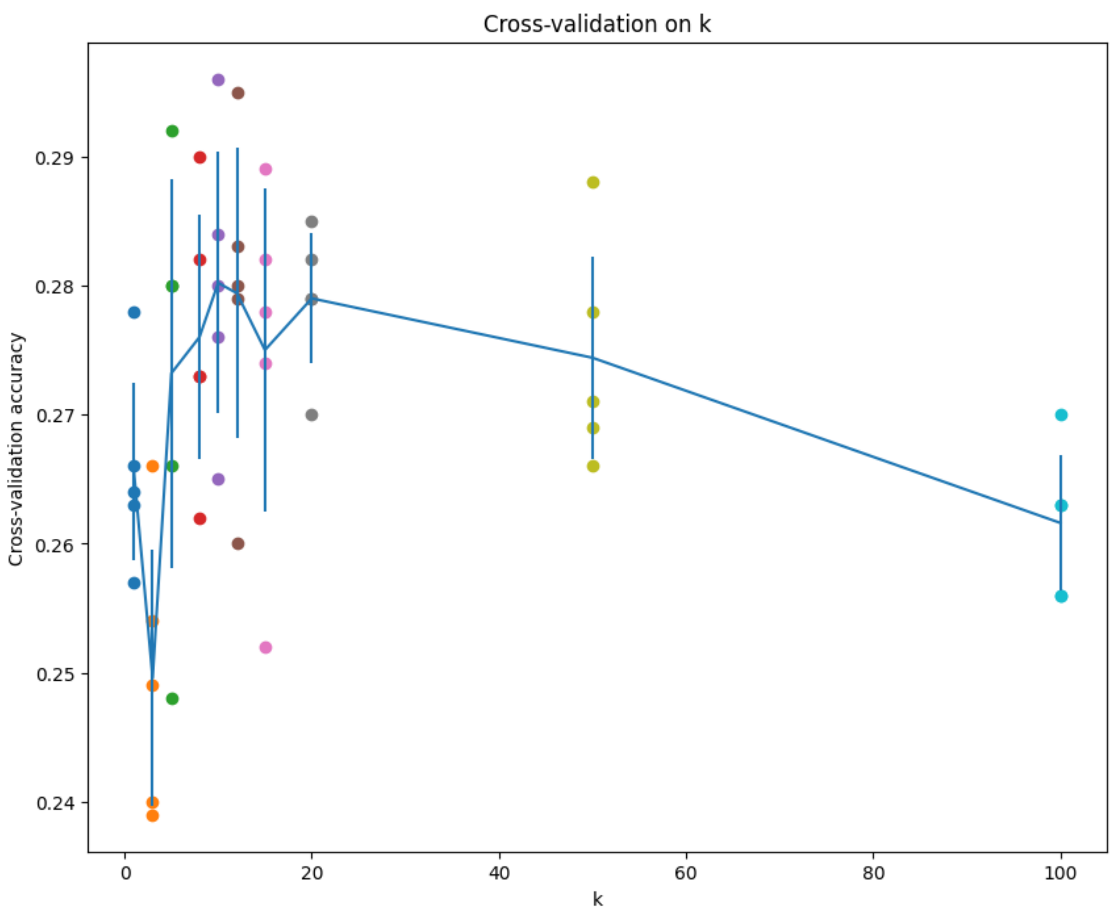
**Inline Question 1**

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

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

*YourAnswer* : a. Bright rows are due to minimal relation between few test images and most of the training images. b. Bright columns are due to minimal relation between few training images and most of the test image.

[10]:
```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
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

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

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

[11]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
# PLEASE DO NOT MODIFY THE MARKERS
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
print('````````````````````````````````````````````````')
```

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

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

**Inline Question 2**

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

the mean $\mu$ across all pixels over all images is

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

And the pixel-wise mean $\mu_{ij}$ across all images is

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

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

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

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

$\color{blue}{\textit{Your Explanation:}}$ 1. Subtracting the mean will retain the values which are above and below the mean point. 2. Subtracting the per pixel mean will normalize the pixel value. 3. Subtracting the mean and dividing by the standard deviation will zero up the mean value of all pixels. 4. But, Subtracting the pixel-wise mean and dividing by the pixel-wise standard deviation will change the scale of all pixels, altering the l1 distance. 5. Rotating the coordinate axes of the data will change the relative distance thus altering the l1 distance.

[12]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# Now lets speed up distance matrix computation by using partial vectorization
```
```

```
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words,␣
↪reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

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

[13]:

```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

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

```
[14]: # PLEASE DO NOT MODIFY THE MARKERS
print('|||||||||||||||||||||||||||||||||||||||||||||||')
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
# PLEASE DO NOT MODIFY THE MARKERS
print('```````````````````````````````````````````````')
```

```
|||||||||||||||||||||||||||||||||||||||||||||||
Two loop version took 43.980442 seconds
One loop version took 70.540304 seconds
No loop version took 0.670533 seconds
```````````````````````````````````````````````
```

### 1.0.1 Cross-validation

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

```
[15]: # PLEASE DO NOT MODIFY THE MARKERS
print('|||||||||||||||||||||||||||||||||||||||||||||||')
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]
```
```

```
X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
# print(y_train_folds[1].shape)

pass

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

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


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for j in range(len(k_choices)):
  accuracy_list = []
  for i in range(num_folds):
    # copying the X_train_folds and y_train_folds list
    new_X_train_folds = X_train_folds.copy()
    new_y_train_folds = y_train_folds.copy()
    # poping the validaion fold from the training data
    X_valid_fold = new_X_train_folds.pop(i)
    y_valid_fold = new_y_train_folds.pop(i)
    # stacking the rest of the training data using
    # np.vstack and np.concatenate
    stacked_X_train_folds = new_X_train_folds[0]
    stacked_y_train_folds = new_y_train_folds[0]
```

```
    for k in range(3):
      stacked_X_train_folds = np.vstack((stacked_X_train_folds,␣
↪new_X_train_folds[k+1]))
      stacked_y_train_folds = np.concatenate((stacked_y_train_folds,␣
↪new_y_train_folds[k+1]))
    # calling the train and distance computing function
    # print(len(stacked_y_train_folds))
    classifier.train(stacked_X_train_folds, stacked_y_train_folds)
    distance = classifier.compute_distances_no_loops(X_valid_fold)
    # predicting the labels according to the distance calculated
    y_valid_pred = classifier.predict_labels(distance, k=k_choices[j])
    # calculating the accuracy percentage
    num_correct = np.sum(y_valid_pred == y_valid_fold)
    # print(y_valid_pred.shape)
    accuracy = float(num_correct) / y_valid_pred.shape[0]
    accuracy_list.append(accuracy)
  # appending the accuracy values of each k
  # in the dictionary
  k_to_accuracies[k_choices[j]] = accuracy_list
pass

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

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

```
||||||||||||||||||||||||||||||||||||||||||||||
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
```

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

[16]:

```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
```

```
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````````')
```

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




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

[17]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||||')
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
```
```

~~~python
best_k = 10

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

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

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

Inline Question 3

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

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

$\color{blue}{Your Explanation:}$ 1. The classification of k-NN is based on the relative distance between the data points. Since there is no assumption that the boundary is linear, this statement is not true. 2. Since during training, 1-NN will always be the same training image itself. Thus the error will be always zero which is lower than 5-NN 3. But in testing, the data is unseen, thus this statement wont be true. 4. Since one of the main assumptions of k-NN is that the data needs to be well sampled. Thus as he dimension increases, the volume of data points increases exponentially. Thus, k-NN needs more time.

svm

September 30, 2024

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

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

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

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

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

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

```
[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
```

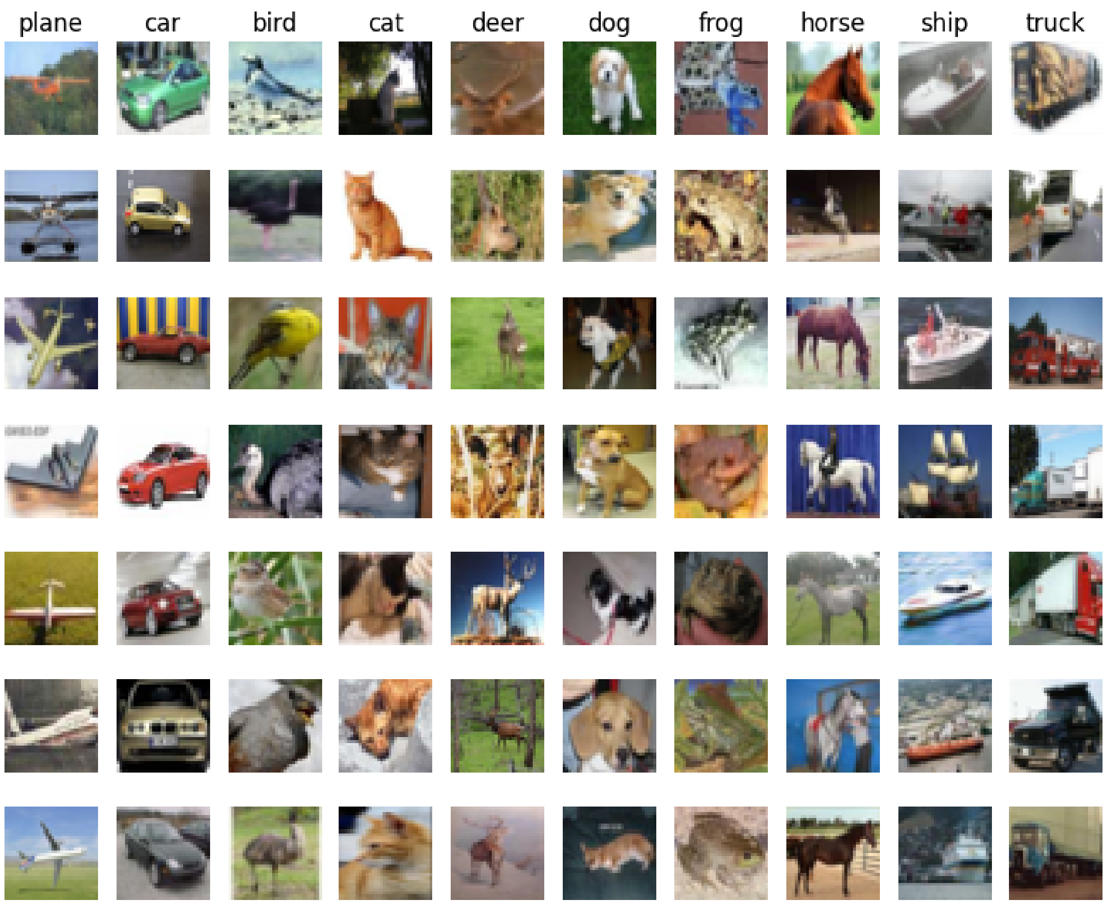
1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

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

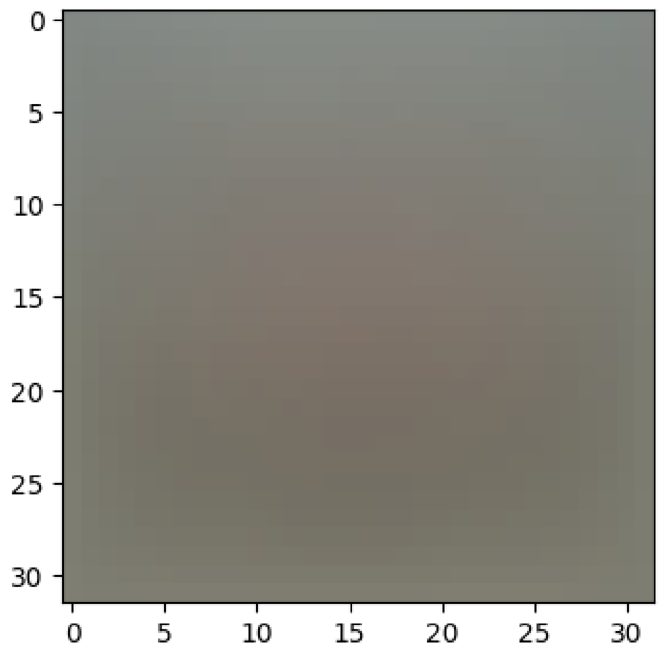
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

```
[7]: # 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.

[8]:
```
# 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: 9.127464
```

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:

[9]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

```
numerical: -27.253216 analytic: -27.253216, relative error: 4.932410e-12
numerical: -7.254698 analytic: -7.254698, relative error: 4.322101e-11
numerical: -41.751061 analytic: -41.751061, relative error: 4.743071e-12
numerical: -3.856000 analytic: -3.856000, relative error: 2.132164e-11
numerical: -35.264322 analytic: -35.264322, relative error: 1.352938e-11
numerical: -0.836598 analytic: -0.778745, relative error: 3.581447e-02
numerical: 7.499644 analytic: 7.499644, relative error: 8.859733e-11
numerical: -19.781632 analytic: -19.781632, relative error: 1.248753e-11
numerical: 9.033119 analytic: 9.102528, relative error: 3.827199e-03
numerical: -11.268685 analytic: -11.268685, relative error: 1.093517e-11
numerical: 3.248087 analytic: 3.167183, relative error: 1.261126e-02
numerical: -10.888196 analytic: -10.888196, relative error: 8.626221e-12
numerical: -5.994749 analytic: -5.994749, relative error: 3.124467e-11
numerical: -15.219401 analytic: -15.219401, relative error: 1.721564e-11
numerical: -6.384204 analytic: -6.384204, relative error: 1.026819e-11
numerical: -31.171255 analytic: -31.166685, relative error: 7.330507e-05
numerical: -30.182566 analytic: -30.182566, relative error: 1.277803e-11
numerical: -32.423836 analytic: -32.423836, relative error: 4.133677e-12
numerical: 3.461850 analytic: 3.461850, relative error: 4.467690e-11
numerical: 29.513383 analytic: 29.513383, relative error: 2.612167e-13
```

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*

$\color{blue}{Your Answer:}$

Yes, this is possible because the hinge loss which is used as the SVM loss function is not differentiable at certain points. Since the probablity of this occuring is very low, its not a reason for concern. For example in Hinge loss, the point s-s_y+1 = 0 is not differentiable. The margin is inversely proporional to this frequency, thus increasing the margin will decrease the frequency of this happening.

[10]:
```
# 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: 9.127464e+00 computed in 0.212161s
Vectorized loss: 9.127464e+00 computed in 0.012604s
difference: 0.000000
```

[11]:
```
# 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')
```

```python
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.222859s
Vectorized loss and gradient: computed in 0.009347s
difference: 0.000000
```

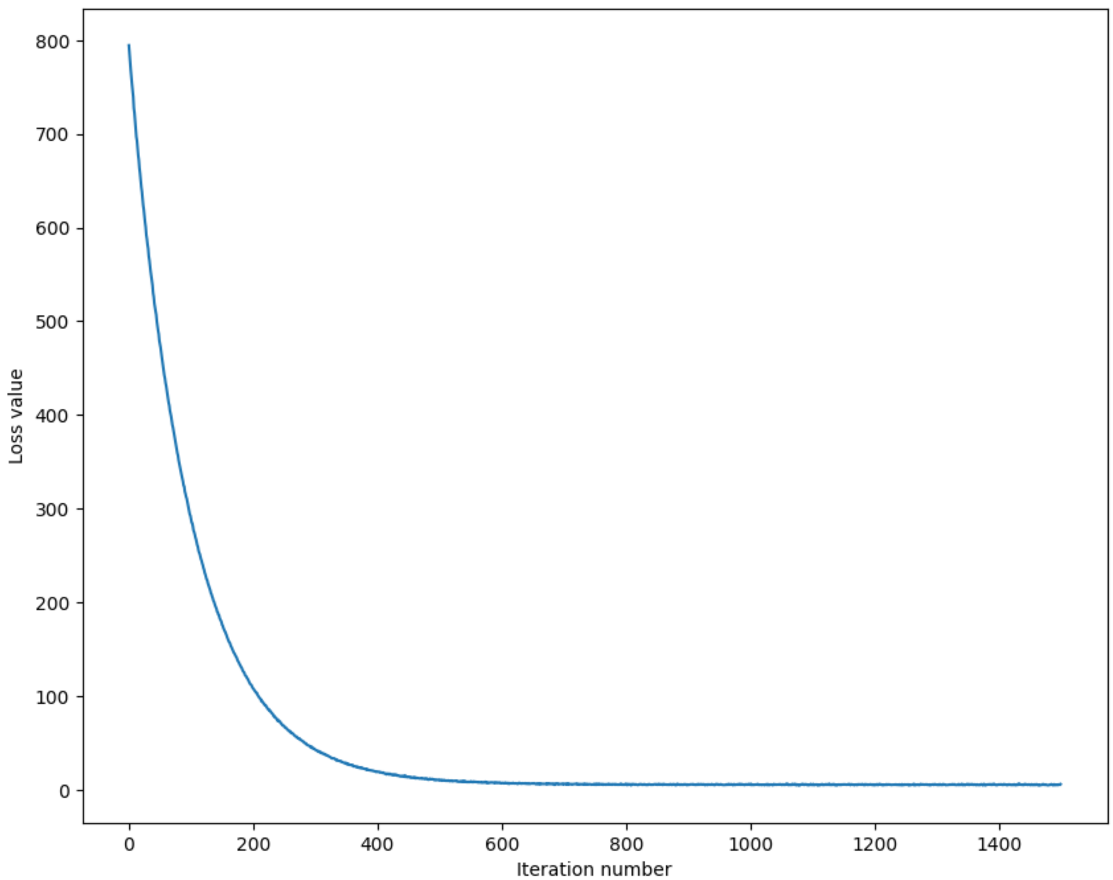
1.2.1 Stochastic Gradient Descent

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

[12]:
```python
# 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 794.440312
iteration 100 / 1500: loss 288.710248
iteration 200 / 1500: loss 107.948612
iteration 300 / 1500: loss 42.399811
iteration 400 / 1500: loss 19.176973
iteration 500 / 1500: loss 9.859566
iteration 600 / 1500: loss 7.132066
iteration 700 / 1500: loss 5.786650
iteration 800 / 1500: loss 6.184682
iteration 900 / 1500: loss 5.277558
iteration 1000 / 1500: loss 5.110897
iteration 1100 / 1500: loss 5.233508
iteration 1200 / 1500: loss 5.569316
iteration 1300 / 1500: loss 5.442515
iteration 1400 / 1500: loss 5.593864
That took 7.356409s
```

[13]:
```python
# 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()
```



[14]:
```
# 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: 0.367490
validation accuracy: 0.376000
```

[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, 5e-7]
regularization_strengths = [2.5e4, 5e4, 1e4, 2e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# itterating through lr and reg list
for i,n in enumerate(learning_rates):
  for j,m in enumerate(regularization_strengths):
    # initializing and traing the SVM
    svm = LinearSVM()
    loss_hist = svm.train(X_train, y_train, learning_rate=n, reg=m,
                          num_iters=1500, verbose=True)
    # predicting and calculating the validation and train accuracies
    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)
    train_accuracy = np.mean(y_train == y_train_pred)
    val_accuracy = np.mean(y_val == y_val_pred)
    # appending the accuracies to the dictionary
    results[(n,m)] = (train_accuracy, val_accuracy)
    # saving the best hyperparameters and SVM object
    if val_accuracy>best_val:
      best_val = val_accuracy
      best_svm = svm
```

```python
# *****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 788.387508
iteration 100 / 1500: loss 286.015125
iteration 200 / 1500: loss 108.441070
iteration 300 / 1500: loss 42.711646
iteration 400 / 1500: loss 18.773844
iteration 500 / 1500: loss 10.543207
iteration 600 / 1500: loss 7.337754
iteration 700 / 1500: loss 5.828006
iteration 800 / 1500: loss 5.943497
iteration 900 / 1500: loss 5.650121
iteration 1000 / 1500: loss 5.629635
iteration 1100 / 1500: loss 4.988943
iteration 1200 / 1500: loss 5.654924
iteration 1300 / 1500: loss 5.443258
iteration 1400 / 1500: loss 5.191402
iteration 0 / 1500: loss 1550.260497
iteration 100 / 1500: loss 209.414870
iteration 200 / 1500: loss 32.570866
iteration 300 / 1500: loss 9.007332
iteration 400 / 1500: loss 6.173800
iteration 500 / 1500: loss 5.622174
iteration 600 / 1500: loss 5.676981
iteration 700 / 1500: loss 5.353188
iteration 800 / 1500: loss 5.189587
iteration 900 / 1500: loss 5.521024
iteration 1000 / 1500: loss 5.981235
iteration 1100 / 1500: loss 5.487942
iteration 1200 / 1500: loss 5.625333
iteration 1300 / 1500: loss 5.724240
iteration 1400 / 1500: loss 5.990787
iteration 0 / 1500: loss 325.428428
iteration 100 / 1500: loss 214.325032
iteration 200 / 1500: loss 144.400044
iteration 300 / 1500: loss 98.068683
iteration 400 / 1500: loss 66.866726
```

```
iteration 500 / 1500: loss 46.511135
iteration 600 / 1500: loss 32.279986
iteration 700 / 1500: loss 23.081629
iteration 800 / 1500: loss 16.779751
iteration 900 / 1500: loss 12.931835
iteration 1000 / 1500: loss 10.195942
iteration 1100 / 1500: loss 8.674028
iteration 1200 / 1500: loss 7.076387
iteration 1300 / 1500: loss 6.146493
iteration 1400 / 1500: loss 5.668403
iteration 0 / 1500: loss 628.199206
iteration 100 / 1500: loss 281.363299
iteration 200 / 1500: loss 127.581315
iteration 300 / 1500: loss 60.047407
iteration 400 / 1500: loss 29.424309
iteration 500 / 1500: loss 16.443718
iteration 600 / 1500: loss 9.652034
iteration 700 / 1500: loss 7.401329
iteration 800 / 1500: loss 6.057653
iteration 900 / 1500: loss 5.787809
iteration 1000 / 1500: loss 5.461645
iteration 1100 / 1500: loss 5.271916
iteration 1200 / 1500: loss 5.050228
iteration 1300 / 1500: loss 5.922468
iteration 1400 / 1500: loss 5.766388
iteration 0 / 1500: loss 790.409444
iteration 100 / 1500: loss 357196174993181014463332569075230441472.000000
iteration 200 / 1500: loss 590416669703627142264704024084589394115061055932918926
06470328237982482432.000000
iteration 300 / 1500: loss 975911469014406066152115852066223134949775010723684661
66033829115519487132467727206761926387397574671393095 68.000000
iteration 400 / 1500: loss 161310349830050685834030549031381098328794006123893816
50697980360428313424111811948204648048255555821858691396636108401191455584684048
91827044352.000000
iteration 500 / 1500: loss 266633089050306259827444968429541250203338104818386750
94454093204381164948665723500700142252082888978832419259228702320828055634875097
942523146912840846763103005766888552184519065 6.000000
iteration 600 / 1500: loss 440723141766223605624289461498324390169364153798234027
85003411230053454161901264633041094629195219591260240169666828674179305971784383
90895900066403552158364875684681657525405279381700854519147471178561306190938112
0.000000
iteration 700 / 1500: loss 728480056170536662554373313812016911054616679548489112
66543391522662188196355435081188113217828044755693304359838187730818680428637299
03659308702737711160767155355862794910961531870062249079745138529843172277220689
26750017789341472797456508314347110 4.000000
iteration 800 / 1500: loss 120411918945641157895773757196949687344418673245930042
15577762323728325423005892552457968969726148509966936437399531312913986686019222
87727625434752407115584613855366579408277277552476565278385606054945753565431 74
```

```
4549390741903151782069795210900806258365665700314368295098059557905104896.000000
```

```
/content/drive/My
Drive/assignment1/assignment1/cs231n/classifiers/linear_svm.py:109:
RuntimeWarning: overflow encountered in scalar multiply
  loss += reg * np.sum(W * W)
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:88:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/drive/My
Drive/assignment1/assignment1/cs231n/classifiers/linear_svm.py:109:
RuntimeWarning: overflow encountered in multiply
  loss += reg * np.sum(W * W)
```

```
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 1571.540406
iteration 100 / 1500: loss 42512596666898581347803183244185178955964922272090219
80895328291424643573774582741325823745597854234341375613093799268450304.000000
iteration 200 / 1500: loss 10977814756025677814266893141679823535812504012673720
88896560644236312254216744802736916640355256716777035760655547157074491503263198
93314215488832700645956860633946450129883069132273379521577069205962997764988798
23941326280212267519736025186304.000000
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf
```

```
/content/drive/My
Drive/assignment1/assignment1/cs231n/classifiers/linear_svm.py:134:
RuntimeWarning: overflow encountered in multiply
  dW += reg * (2 * W)
```

```
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 331.036689
iteration 100 / 1500: loss 535.192573
iteration 200 / 1500: loss 542.214277
iteration 300 / 1500: loss 643.634398
iteration 400 / 1500: loss 695.106951
```

```
iteration 500 / 1500: loss 569.476712
iteration 600 / 1500: loss 715.943903
iteration 700 / 1500: loss 614.814463
iteration 800 / 1500: loss 554.223180
iteration 900 / 1500: loss 486.948816
iteration 1000 / 1500: loss 561.208552
iteration 1100 / 1500: loss 614.911349
iteration 1200 / 1500: loss 507.876881
iteration 1300 / 1500: loss 656.072032
iteration 1400 / 1500: loss 597.888375
iteration 0 / 1500: loss 640.127295
iteration 100 / 1500: loss 4156189.050208
iteration 200 / 1500: loss 16577691.194554
iteration 300 / 1500: loss 37333246.686571
iteration 400 / 1500: loss 66481071.098363
iteration 500 / 1500: loss 104300298.641982
iteration 600 / 1500: loss 150038233.162826
iteration 700 / 1500: loss 204226726.975236
iteration 800 / 1500: loss 266963779.111725
iteration 900 / 1500: loss 337791092.369311
iteration 1000 / 1500: loss 415102424.738971
iteration 1100 / 1500: loss 500790297.689250
iteration 1200 / 1500: loss 594319879.735396
iteration 1300 / 1500: loss 700002490.149411
iteration 1400 / 1500: loss 813477932.758165
iteration 0 / 1500: loss 782.366025
iteration 100 / 1500: loss 10.418595
iteration 200 / 1500: loss 5.608798
iteration 300 / 1500: loss 5.958173
iteration 400 / 1500: loss 5.339843
iteration 500 / 1500: loss 5.529330
iteration 600 / 1500: loss 6.104401
iteration 700 / 1500: loss 5.421054
iteration 800 / 1500: loss 5.745413
iteration 900 / 1500: loss 5.963424
iteration 1000 / 1500: loss 5.999125
iteration 1100 / 1500: loss 6.363579
iteration 1200 / 1500: loss 5.638727
iteration 1300 / 1500: loss 6.172441
iteration 1400 / 1500: loss 5.726915
iteration 0 / 1500: loss 1545.418131
iteration 100 / 1500: loss 6.777176
iteration 200 / 1500: loss 5.785460
iteration 300 / 1500: loss 6.547850
iteration 400 / 1500: loss 5.753107
iteration 500 / 1500: loss 5.998099
iteration 600 / 1500: loss 6.176540
iteration 700 / 1500: loss 6.261242
```

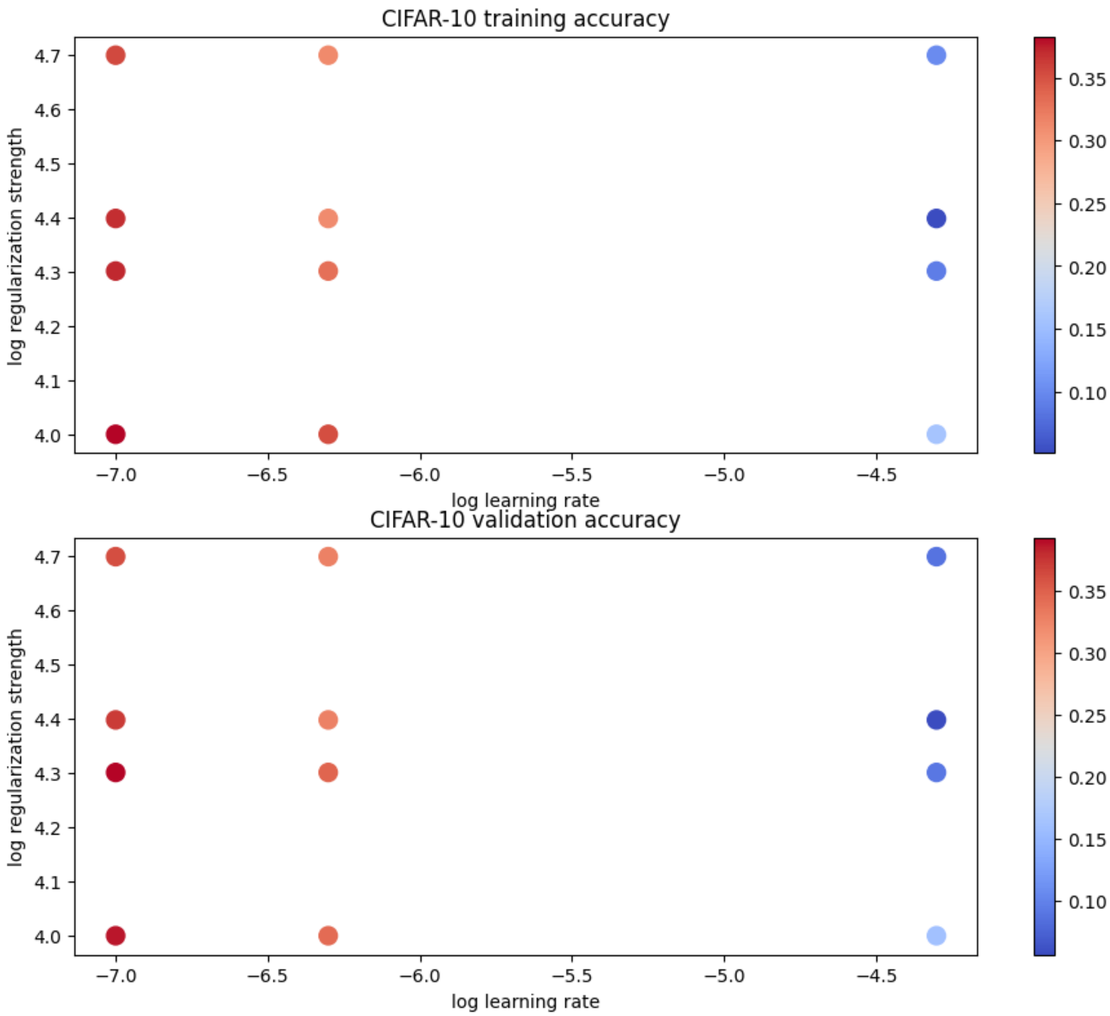
```
iteration 800 / 1500: loss 6.168359
iteration 900 / 1500: loss 5.949905
iteration 1000 / 1500: loss 5.474327
iteration 1100 / 1500: loss 5.884044
iteration 1200 / 1500: loss 5.726960
iteration 1300 / 1500: loss 6.060684
iteration 1400 / 1500: loss 5.655330
iteration 0 / 1500: loss 329.458286
iteration 100 / 1500: loss 46.092556
iteration 200 / 1500: loss 10.854371
iteration 300 / 1500: loss 5.604734
iteration 400 / 1500: loss 5.439480
iteration 500 / 1500: loss 5.783092
iteration 600 / 1500: loss 4.877676
iteration 700 / 1500: loss 4.568482
iteration 800 / 1500: loss 5.209680
iteration 900 / 1500: loss 5.164676
iteration 1000 / 1500: loss 5.437195
iteration 1100 / 1500: loss 5.087190
iteration 1200 / 1500: loss 5.501940
iteration 1300 / 1500: loss 5.642774
iteration 1400 / 1500: loss 5.627584
iteration 0 / 1500: loss 632.956575
iteration 100 / 1500: loss 16.142523
iteration 200 / 1500: loss 5.403013
iteration 300 / 1500: loss 4.743788
iteration 400 / 1500: loss 6.513523
iteration 500 / 1500: loss 5.750038
iteration 600 / 1500: loss 5.634195
iteration 700 / 1500: loss 5.087052
iteration 800 / 1500: loss 5.655411
iteration 900 / 1500: loss 5.011623
iteration 1000 / 1500: loss 5.168264
iteration 1100 / 1500: loss 5.680852
iteration 1200 / 1500: loss 5.450088
iteration 1300 / 1500: loss 5.748058
iteration 1400 / 1500: loss 5.641813
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.382918 val accuracy: 0.387000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.371490 val accuracy: 0.393000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.368694 val accuracy: 0.372000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.355388 val accuracy: 0.362000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.352939 val accuracy: 0.343000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.330082 val accuracy: 0.347000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.311082 val accuracy: 0.328000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.312796 val accuracy: 0.328000
lr 5.000000e-05 reg 1.000000e+04 train accuracy: 0.157959 val accuracy: 0.161000
lr 5.000000e-05 reg 2.000000e+04 train accuracy: 0.089306 val accuracy: 0.089000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.050918 val accuracy: 0.056000
```

```
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.393000
```

[16]:

```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()
print(results)
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()
```

```
{(1e-07, 25000.0): (0.3686938775510204, 0.372), (1e-07, 50000.0):
(0.3553877551020408, 0.362), (1e-07, 10000.0): (0.3829183673469388, 0.387),
(1e-07, 20000.0): (0.37148979591836734, 0.393), (5e-05, 25000.0):
(0.050918367346938775, 0.056), (5e-05, 50000.0): (0.10026530612244898, 0.087),
(5e-05, 10000.0): (0.1579591836734694, 0.161), (5e-05, 20000.0):
(0.08930612244897959, 0.089), (5e-07, 25000.0): (0.3110816326530612, 0.328),
(5e-07, 50000.0): (0.31279591836734694, 0.328), (5e-07, 10000.0):
(0.3529387755102041, 0.343), (5e-07, 20000.0): (0.33008163265306123, 0.347)}
```

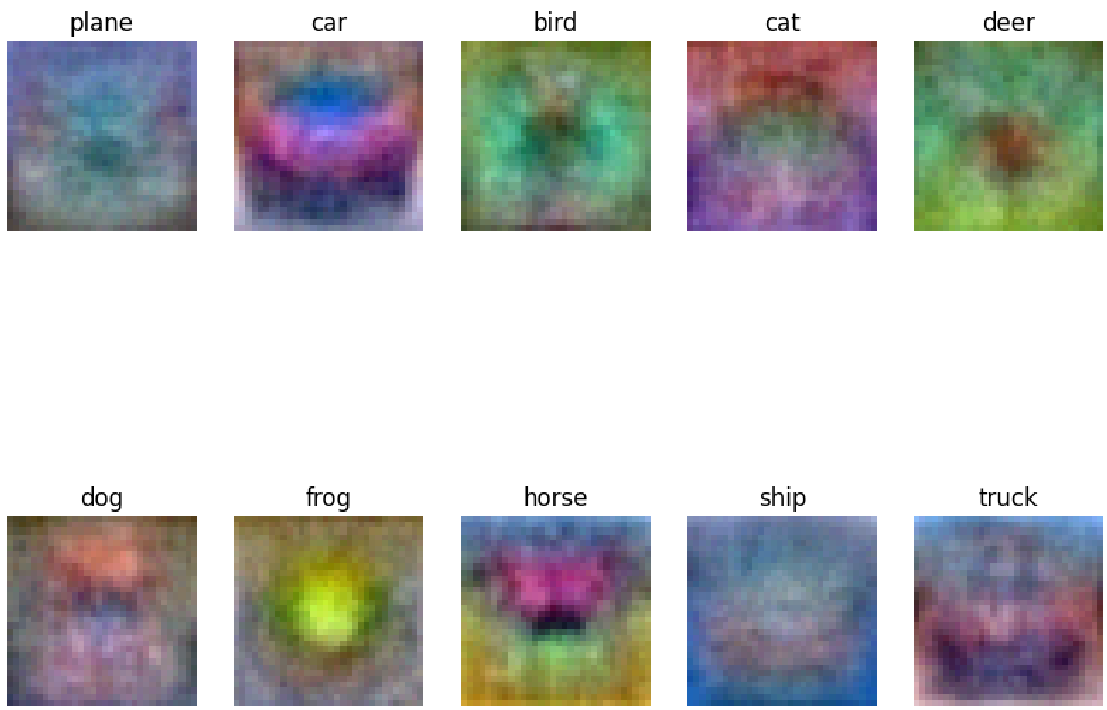


[17]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.366000
```

[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 : When the weights are reshaped and transformed in RGB images, it resembles the visual characters of each class. These images look like a blurry and noisy version of the class images. This happens because the weights are optimized inorder to represent the key features of the class.

softmax

September 30, 2024

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

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

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

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

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

1 Softmax exercise

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

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

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

[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: 2.383482
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer : Since at initialization stage, the model does not know about the input features, the output becomes the baseline probability. Which in this case is about 10 percentage. Thus, the value is closer to -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: -1.034883 analytic: -1.034883, relative error: 1.131313e-08
numerical: 1.257779 analytic: 1.257778, relative error: 2.543872e-08
numerical: -0.828274 analytic: -0.828274, relative error: 2.714631e-08
numerical: -1.677397 analytic: -1.677397, relative error: 4.119118e-09
numerical: -1.009414 analytic: -1.009414, relative error: 4.324014e-08
```

```
numerical: -0.314061 analytic: -0.314061, relative error: 2.615106e-08
numerical: 3.950168 analytic: 3.950168, relative error: 2.888094e-08
numerical: -3.388142 analytic: -3.388142, relative error: 6.846439e-09
numerical: -0.521788 analytic: -0.521788, relative error: 2.600687e-08
numerical: 0.572668 analytic: 0.572668, relative error: 1.553945e-08
numerical: 1.016270 analytic: 1.016269, relative error: 8.235284e-08
numerical: 1.856948 analytic: 1.856948, relative error: 4.489018e-08
numerical: -0.544846 analytic: -0.544846, relative error: 4.760286e-08
numerical: -4.264841 analytic: -4.264841, relative error: 9.833716e-09
numerical: -0.815226 analytic: -0.815226, relative error: 1.194702e-07
numerical: 0.948043 analytic: 0.948043, relative error: 5.264985e-10
numerical: -1.413382 analytic: -1.413382, relative error: 9.897088e-09
numerical: -0.034107 analytic: -0.034107, relative error: 1.570905e-06
numerical: 0.946253 analytic: 0.946253, relative error: 2.747453e-08
numerical: 0.476496 analytic: 0.476496, relative error: 1.420331e-07
```

[6]:

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

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

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

```
naive loss: 2.383482e+00 computed in 0.133691s
vectorized loss: 2.383482e+00 computed in 0.044216s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[10]:

```python
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
```

```python
# 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]
regularization_strengths = [2.5e4, 5e4]

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

for i in learning_rates:
  for j in regularization_strengths:
    # initializing and traing the SVM
    softmax = Softmax()
    loss_hist = softmax.train(X_train, y_train, learning_rate=i, reg=j,
                      num_iters=1500, verbose=True)
    # predicting and calculating the validation and train accuracies
    y_train_pred = softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)
    train_accuracy = np.mean(y_train == y_train_pred)
    val_accuracy = np.mean(y_val == y_val_pred)
    # appending the accuracies to the dictionary
    results[(i,j)] = (train_accuracy, val_accuracy)
    # saving the best hyperparameters and SVM object
    if val_accuracy>best_val:
      best_val = val_accuracy
      best_softmax = softmax

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

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

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

```
iteration 0 / 1500: loss 783.267171
iteration 100 / 1500: loss 287.742070
iteration 200 / 1500: loss 106.549169
iteration 300 / 1500: loss 40.321733
iteration 400 / 1500: loss 16.153462
iteration 500 / 1500: loss 7.233281
iteration 600 / 1500: loss 3.904957
iteration 700 / 1500: loss 2.783851
iteration 800 / 1500: loss 2.358776
iteration 900 / 1500: loss 2.140393
iteration 1000 / 1500: loss 2.208035
iteration 1100 / 1500: loss 2.090943
iteration 1200 / 1500: loss 2.081401
iteration 1300 / 1500: loss 2.117098
iteration 1400 / 1500: loss 2.108486
iteration 0 / 1500: loss 1533.470098
iteration 100 / 1500: loss 206.386654
iteration 200 / 1500: loss 29.384672
iteration 300 / 1500: loss 5.802447
iteration 400 / 1500: loss 2.630874
iteration 500 / 1500: loss 2.239090
iteration 600 / 1500: loss 2.150913
iteration 700 / 1500: loss 2.150961
iteration 800 / 1500: loss 2.168744
iteration 900 / 1500: loss 2.163141
iteration 1000 / 1500: loss 2.122168
iteration 1100 / 1500: loss 2.159744
iteration 1200 / 1500: loss 2.145240
iteration 1300 / 1500: loss 2.138468
iteration 1400 / 1500: loss 2.144916
iteration 0 / 1500: loss 773.660233
iteration 100 / 1500: loss 6.948892
iteration 200 / 1500: loss 2.157645
iteration 300 / 1500: loss 2.015806
iteration 400 / 1500: loss 2.107826
iteration 500 / 1500: loss 2.090115
iteration 600 / 1500: loss 2.057119
iteration 700 / 1500: loss 2.099468
iteration 800 / 1500: loss 2.090095
iteration 900 / 1500: loss 2.065483
iteration 1000 / 1500: loss 2.043716
iteration 1100 / 1500: loss 2.068541
iteration 1200 / 1500: loss 2.109615
```

```
iteration 1300 / 1500: loss 2.073852
iteration 1400 / 1500: loss 2.136810
iteration 0 / 1500: loss 1536.128348
iteration 100 / 1500: loss 2.162662
iteration 200 / 1500: loss 2.110774
iteration 300 / 1500: loss 2.123293
iteration 400 / 1500: loss 2.182824
iteration 500 / 1500: loss 2.167912
iteration 600 / 1500: loss 2.130848
iteration 700 / 1500: loss 2.099589
iteration 800 / 1500: loss 2.131738
iteration 900 / 1500: loss 2.148782
iteration 1000 / 1500: loss 2.122884
iteration 1100 / 1500: loss 2.199132
iteration 1200 / 1500: loss 2.159248
iteration 1300 / 1500: loss 2.179697
iteration 1400 / 1500: loss 2.178822
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329551 val accuracy: 0.339000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.314959 val accuracy: 0.329000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.334857 val accuracy: 0.353000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.303286 val accuracy: 0.321000
best validation accuracy achieved during cross-validation: 0.353000
```

[11]:
```
# 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: 0.343000
```

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.

$\color{blue}{\textit{Your Answer :}}$ Yes, SVM loss may tend to remain unchanged, while Softmax changes.

$\color{blue}{\textit{Your Explanation :}}$ For example, if the score of the correct class changes from 20 to 30. SVM loss will remain unchanged. On other hand, the probablity of that particular class in Softmax increases. It is because Softmax will make minimal changes until the probability reaches 1.

[12]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

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

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

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

[ ]:

```
```

two_layer_net

September 30, 2024

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

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

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

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

1 Fully-Connected Neural Nets

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

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

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

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

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

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

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

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

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

2 Affine layer: forward

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

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

[ ]:
```
# Test the affine_forward function

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

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

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

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

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

3 Affine layer: backward

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

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

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

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

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

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

4 ReLU activation: forward

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

[ ]:
```
# Test the relu_forward function

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

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

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

5 ReLU activation: backward

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

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

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

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

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

5.1 Inline Question 1:

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

5.2 Answer:

[FILL THIS IN]

6 "Sandwich" layers

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

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

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

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

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

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

7 Loss layers: Softmax and SVM

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

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

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

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

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

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

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

8 Two-layer network

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

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

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

print('Testing initialization ... ')
```

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

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

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

9 Solver

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

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

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

pass

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

10 Debug the training

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

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

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

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

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
```

```
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```

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

# Visualize the weights of the network

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

show_net_weights(model)
```

11 Tune your hyperparameters

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

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

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

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

[ ]:
```
best_model = None


##################################################################################
```

```
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
 ↪#
# model in best_model.                                                          ␣
 ↪#
#                                                                               ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative    ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.          ␣
 ↪#
#                                                                               ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters          ␣
 ↪#
# automatically like we did on thexs previous exercises.                        ␣
 ↪  #
##################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

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

12 Test your model!

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

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

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

12.1 Inline Question 2:

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

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

$\textit{YourAnswer}:$

$\textit{YourExplanation}:$

```
[ ]:
```

features

September 30, 2024

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

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

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

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

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.

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

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

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

1.1 Load data

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

[ ]:
```
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.

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

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.

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

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1
best_svm = None

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

pass

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

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

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

[ ]:
```
# 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)
```

[ ]:
```
# 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?

$\color{blue}{\textit{Your Answer :}}$

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.

[ ]:
```
# 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)
```

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

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

```python
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)*****

pass

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

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

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