knn

January 22, 2024

[18]:
```python
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cse493g1/assignments/assignment1/'
FOLDERNAME = 'cse493g1/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignments/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

[19]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

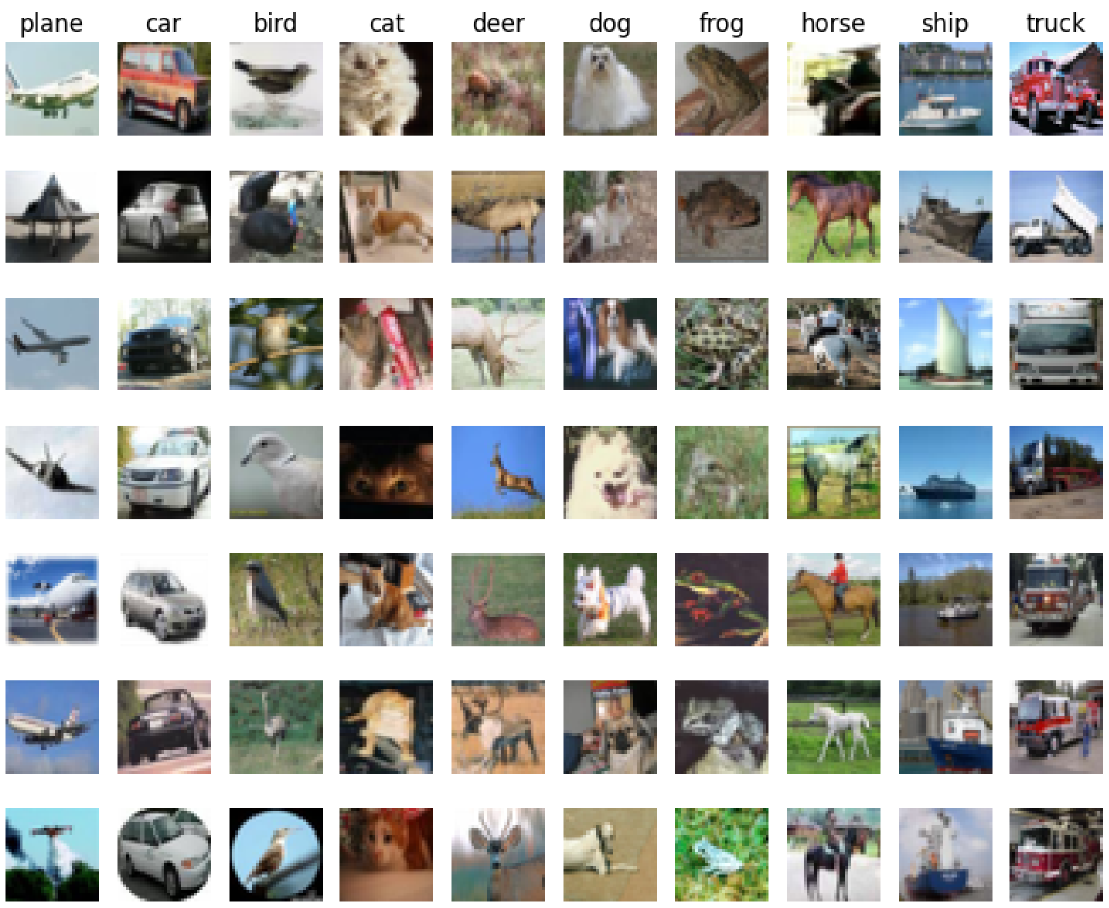
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[20]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cse493g1/datasets/cifar-10-batches-py'

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

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

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

```
Clear previously loaded data.
```

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

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



[22]:

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

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

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

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

[23]:
```python
from cse493g1.classifiers import KNearestNeighbor

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

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

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

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

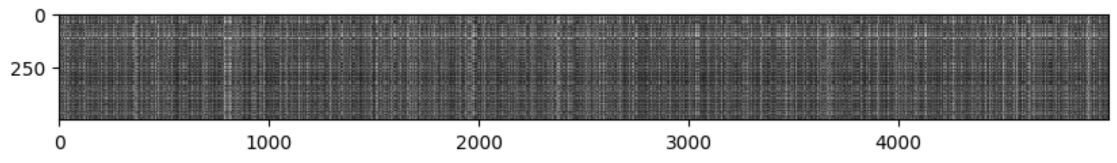
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cse493g1/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.

[24]:
```python
# Open cse493g1/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

[25]:
```python
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

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

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

$\color{blue}{Your Answer:}$ *fill this in.*

- Distinctly bright rows (i.e. rows which are mostly white) could be caused by images in the test set that are very dissimilar to the training dataset. This could be due to unique pixel structures given our representation of images.

- Distincly bright columns could be caused by images in the training dataset which are very dissimilar to the test dataset. Again, this could be due to unique pixel structures in our training images

[26]:
```python
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

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

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

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

[27]:
```python
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

$Your Answer:$

1,2 and 3

$Your Explanation:$

Going case by case

1. Subtracting the mean from all pixels will not change L1 distances between them as it will cancel out, hence it won't change the L1 norm

2. Subtracting the per pixel mean will also not change the L1 distances as the difference between each pixel will be calculated seperately and then summed up. The pixel wise absolute difference won't change as the pixel wise mean will cancel out.

3. Subtracting by mean and standard deviation will change L1 distances, howevere they will only be scaled by $1/\sigma$ across the board. This means that while the distances are changed, k nearest neighbours or the k shortest distances to a test point will remain the same. Hence the algorithm won't change it's classification for any point

4. Subtracting by pixel wise mean and dividing by the pixel wise standard deviation will change the L1 distances and not in a uniform manner across data points. Hence, the performance of the classifier will change

5. Rotating the coordinate axis changes L1 distances and not in a uniform manner. Hence, the performance of teh classifier will change

[28]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

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

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

[29]:
```python
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

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

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

[30]:
```python
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took␣
 ↪to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

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

```
Two loop version took 41.235950 seconds
One loop version took 51.590850 seconds
No loop version took 0.605624 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.

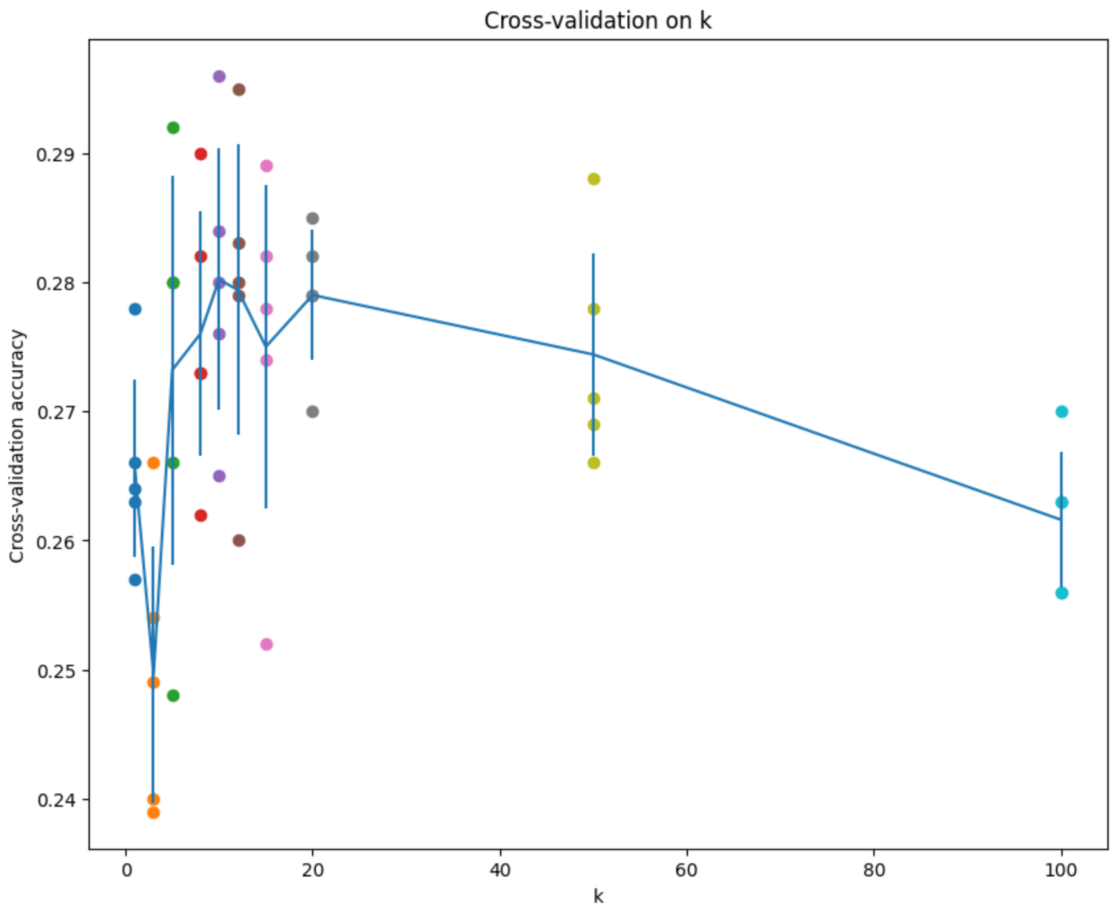
[31]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)

# *****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)*****

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

for i in range(num_folds):
  #Create test set for cross validation
  X_test_crossval = X_train_folds[i]
  y_test_crossval = y_train_folds[i]

  #Create training set for cross validation
  X_train_crossval = np.concatenate([X_train_folds[j] for j in range(num_folds)␣
↪if j!=i])
  y_train_crossval = np.concatenate([y_train_folds[j] for j in range(num_folds)␣
↪if j!=i])

  #Train model and calculate distance matrix
  classifier_crossval = KNearestNeighbor()
  classifier_crossval.train(X_train_crossval, y_train_crossval)
  dists_crossval = classifier_crossval.
↪compute_distances_no_loops(X_test_crossval)

  #Calculate accuracies for given values of k
  for k in k_choices:
    y_test_pred_crossval = classifier_crossval.predict_labels(dists_crossval, k)
    num_correct = np.sum(y_test_pred_crossval == y_test_crossval)
    num_test_crossval = y_test_crossval.shape[0]
    accuracy = float(num_correct) / num_test_crossval
    k_to_accuracies[k].append(accuracy)

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

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

```
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
```

[32]:

```
# 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()
```



[33]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than 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.

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

2 and 4

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

Decision boundary of k-NN isn't necessarily linear as we saw in class.

Training error of a 1-NN is always zero as it perfectly fits the training data, hence statement 2 holds. However this doesn't guarantee performance on test data, making 3 untrue

Classification with k-NN requires computation of the distance matrix which increases in complexity as the training set grows, hence 4 is also correct.

svm

January 22, 2024

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

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cse493g1/assignments/assignment1/'
FOLDERNAME = 'cse493g1/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignments/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment1
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

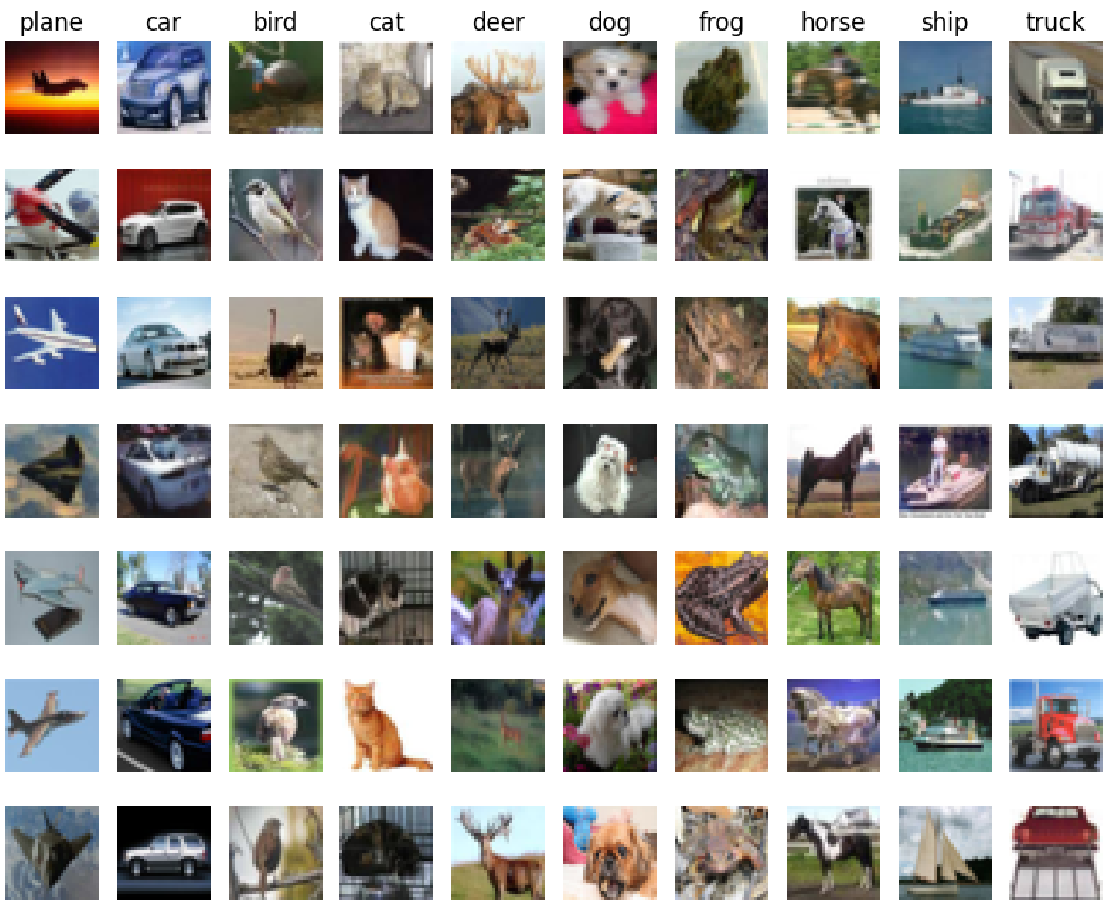
[2]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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 = 'cse493g1/datasets/cifar-10-batches-py'

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

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

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

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

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



[5]:

```python
# 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)
```

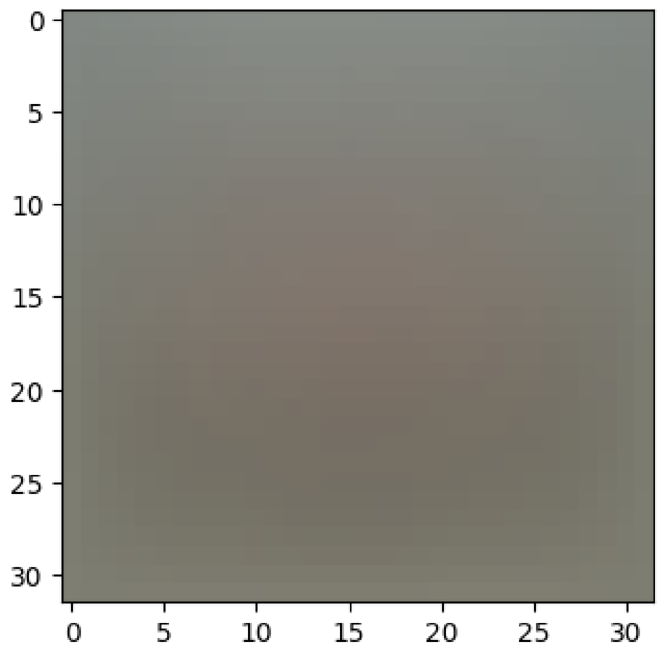
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[6]:
```python
# 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]:
```python
# 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 `cse493g1/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]:
```python
# Evaluate the naive implementation of the loss we provided for you:
from cse493g1.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 8.726496
```

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, 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 cse493g1.gradient_check import grad_check_sparse
     f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
     grad_numerical = grad_check_sparse(f, W, grad)

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

```
numerical: -19.172319 analytic: -19.172319, relative error: 1.190506e-11
numerical: -18.399764 analytic: -18.399764, relative error: 8.082238e-12
numerical: -3.788839 analytic: -3.788839, relative error: 3.650172e-11
numerical: -14.254513 analytic: -14.254513, relative error: 1.233461e-12
numerical: 44.418277 analytic: 44.418277, relative error: 1.078973e-12
numerical: -3.550071 analytic: -3.550071, relative error: 8.029304e-11
numerical: 18.944904 analytic: 18.944904, relative error: 4.505565e-12
numerical: -39.311086 analytic: -39.311086, relative error: 1.947929e-12
numerical: -44.697694 analytic: -44.697694, relative error: 2.744235e-12
numerical: -41.369419 analytic: -41.369419, relative error: 7.307598e-12
numerical: -7.517842 analytic: -7.517842, relative error: 3.916432e-11
numerical: -7.172637 analytic: -7.172637, relative error: 4.187033e-11
numerical: -7.765225 analytic: -7.765225, relative error: 2.929394e-11
numerical: 22.775897 analytic: 22.775897, relative error: 4.881416e-12
numerical: 4.106511 analytic: 4.106511, relative error: 1.160836e-11
numerical: 4.870183 analytic: 4.870183, relative error: 1.390503e-11
numerical: -20.280257 analytic: -20.280257, relative error: 4.160546e-14
numerical: 5.202513 analytic: 5.202513, relative error: 2.689932e-11
numerical: -1.579506 analytic: -1.579506, relative error: 2.199524e-10
numerical: 7.041288 analytic: 7.041288, relative error: 1.319978e-12
```

Inline Question 1

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

Your Answer : Yes, numerical gradients can have incorrect dimensions if loss function values are very close to zero. This would be because SVM loss is not differentiable at 0. This means for a value of x in the gradcheck function like say -0.5 and an increment h of say 1, the gradcheck function would return (0.5-0)/2 = 0.025. This is incorrect as we know the gradient for all negative values should be 0.

We can reduce the effect of this by choosing smaller values of h such that the intervals chosen by the gradcheck function do not cross the point of non-differentiability.

[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 cse493g1.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
Naive loss: 8.726496e+00 computed in 0.146493s
Vectorized loss: 8.726496e+00 computed in 0.020512s
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')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.129489s
Vectorized loss and gradient: computed in 0.017418s
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 `cse493g1/classifiers/linear_classifier.py`.

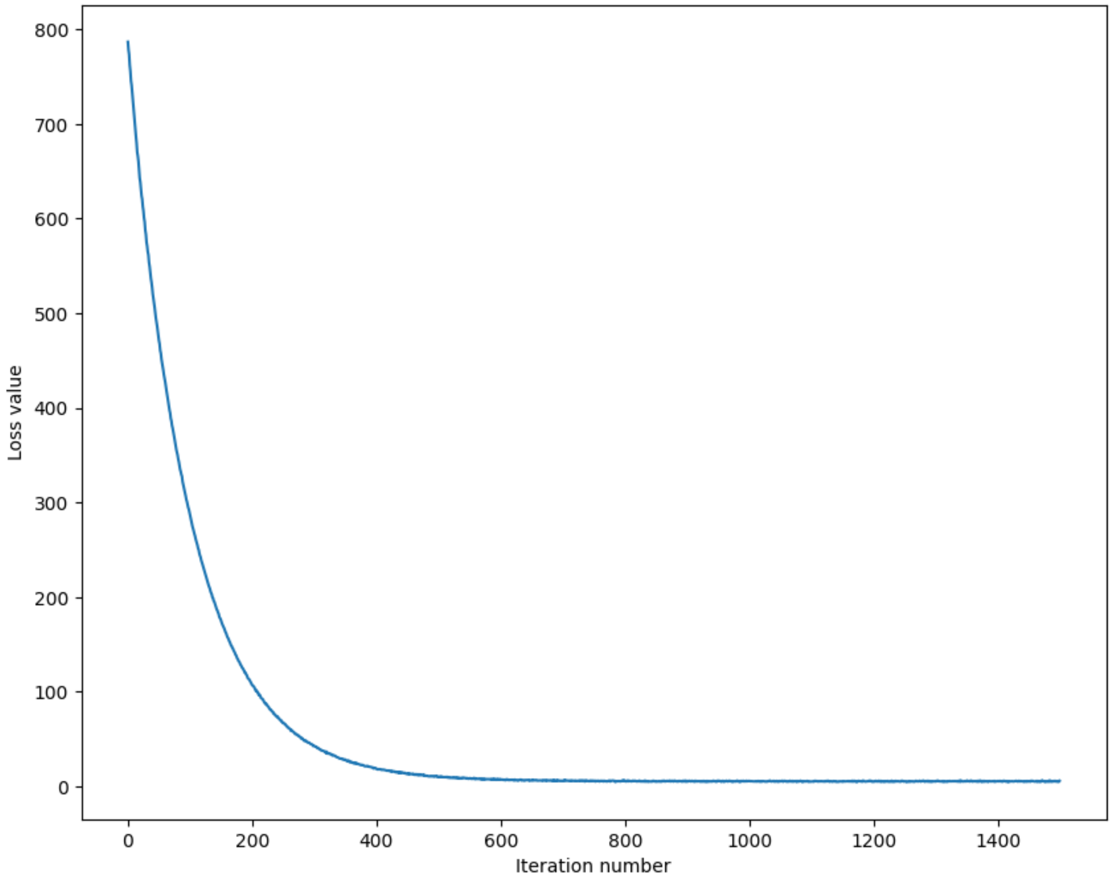
[12]:
```
batch = np.random.choice(100,10,replace = False)
print(batch.shape)
print(batch)
```

```
(10,)
[24 31 19 38 63 60 65 96 16 70]
```

[13]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cse493g1.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 786.843575
iteration 100 / 1500: loss 286.894831
iteration 200 / 1500: loss 106.844766
iteration 300 / 1500: loss 42.600350
iteration 400 / 1500: loss 18.437607
iteration 500 / 1500: loss 9.773369
iteration 600 / 1500: loss 7.369793
iteration 700 / 1500: loss 5.664577
iteration 800 / 1500: loss 5.407622
iteration 900 / 1500: loss 4.930913
iteration 1000 / 1500: loss 5.797234
iteration 1100 / 1500: loss 5.234251
iteration 1200 / 1500: loss 5.378014
iteration 1300 / 1500: loss 5.141340
iteration 1400 / 1500: loss 5.679101
That took 11.763444s
```

[14]:
```
# 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()
```



[15]:
```
# 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), ))
```

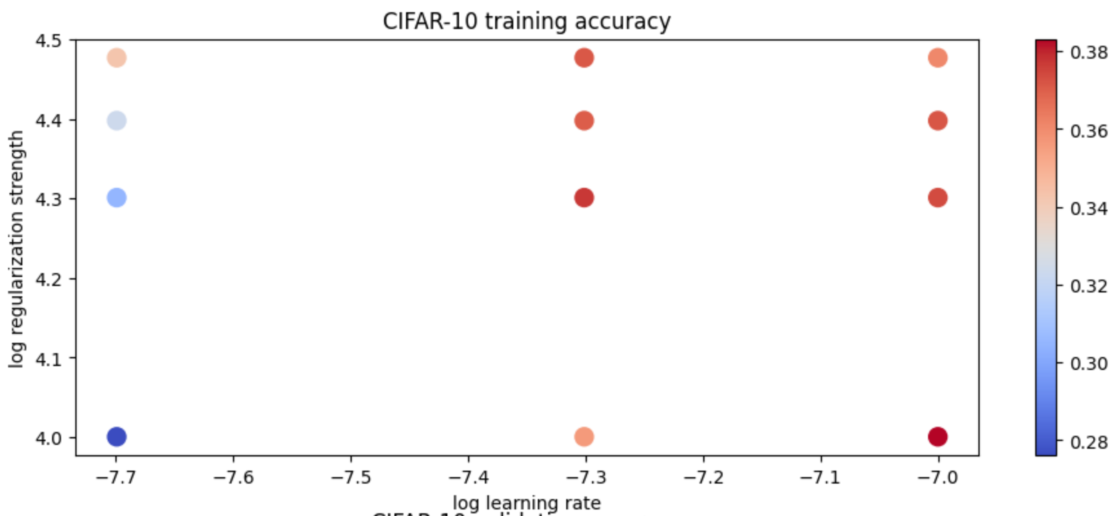
```
training accuracy: 0.370959
validation accuracy: 0.373000
```

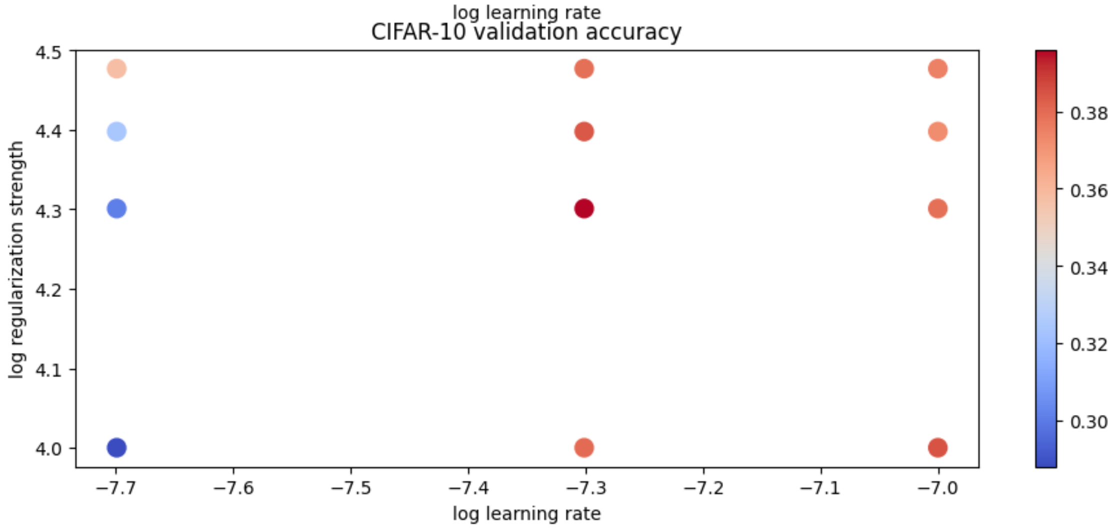
[16]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) 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]
#regularization_strengths = [2.5e4, 5e4]
learning_rates = [1e-7, 0.5e-7,0.5e-7, 0.2e-7]
regularization_strengths = [1e4, 2e4, 2.5e4, 3e4]

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

for learn_rate in learning_rates:
  for reg_strength in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate= learn_rate, reg=reg_strength,␣
 ↪num_iters=1500, verbose=True)

    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)

    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm

    results[(learn_rate,reg_strength)] = (train_accuracy, val_accuracy)


# *****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 327.834390
iteration 100 / 1500: loss 214.666563
iteration 200 / 1500: loss 143.929930
iteration 300 / 1500: loss 97.384635
iteration 400 / 1500: loss 66.123329
iteration 500 / 1500: loss 46.186480
iteration 600 / 1500: loss 32.268340
iteration 700 / 1500: loss 23.346334
iteration 800 / 1500: loss 17.515144
iteration 900 / 1500: loss 12.877809
iteration 1000 / 1500: loss 10.182003
iteration 1100 / 1500: loss 8.625433
iteration 1200 / 1500: loss 7.303914
iteration 1300 / 1500: loss 6.482888
iteration 1400 / 1500: loss 5.683078
iteration 0 / 1500: loss 635.049025
iteration 100 / 1500: loss 282.702284
iteration 200 / 1500: loss 128.490071
iteration 300 / 1500: loss 59.735814
iteration 400 / 1500: loss 29.175947
iteration 500 / 1500: loss 15.624179
iteration 600 / 1500: loss 9.886169
iteration 700 / 1500: loss 6.798961
iteration 800 / 1500: loss 6.603370
iteration 900 / 1500: loss 5.366406
iteration 1000 / 1500: loss 5.148430
iteration 1100 / 1500: loss 5.350315
iteration 1200 / 1500: loss 5.380012
iteration 1300 / 1500: loss 5.113572
iteration 1400 / 1500: loss 5.470066
iteration 0 / 1500: loss 788.450448
iteration 100 / 1500: loss 287.091013
iteration 200 / 1500: loss 107.623234
iteration 300 / 1500: loss 43.047335
iteration 400 / 1500: loss 19.037474
iteration 500 / 1500: loss 10.585243
iteration 600 / 1500: loss 6.743919
iteration 700 / 1500: loss 6.152034
iteration 800 / 1500: loss 5.854667
iteration 900 / 1500: loss 5.710004
iteration 1000 / 1500: loss 4.913486
iteration 1100 / 1500: loss 4.947059
iteration 1200 / 1500: loss 5.580709
iteration 1300 / 1500: loss 5.197128
iteration 1400 / 1500: loss 5.709298
iteration 0 / 1500: loss 945.305602
iteration 100 / 1500: loss 282.351793
iteration 200 / 1500: loss 87.537358
iteration 300 / 1500: loss 29.905140
iteration 400 / 1500: loss 12.555619
iteration 500 / 1500: loss 7.129204
iteration 600 / 1500: loss 6.035207
iteration 700 / 1500: loss 5.616022
iteration 800 / 1500: loss 5.329920
iteration 900 / 1500: loss 5.397303
iteration 1000 / 1500: loss 5.641697
iteration 1100 / 1500: loss 5.295281
iteration 1200 / 1500: loss 5.464028
iteration 1300 / 1500: loss 5.798594
iteration 1400 / 1500: loss 5.411713
iteration 0 / 1500: loss 328.090500
iteration 100 / 1500: loss 260.051023
iteration 200 / 1500: loss 212.270582
iteration 300 / 1500: loss 176.160176
iteration 400 / 1500: loss 142.192616
iteration 500 / 1500: loss 117.521503
iteration 600 / 1500: loss 97.589780
iteration 700 / 1500: loss 80.503681
iteration 800 / 1500: loss 66.212633
iteration 900 / 1500: loss 54.935695
iteration 1000 / 1500: loss 46.183316
iteration 1100 / 1500: loss 38.625106
iteration 1200 / 1500: loss 32.016333
iteration 1300 / 1500: loss 26.283082
iteration 1400 / 1500: loss 22.762463
iteration 0 / 1500: loss 632.649414
iteration 100 / 1500: loss 417.923992
iteration 200 / 1500: loss 280.823775
iteration 300 / 1500: loss 188.572295
iteration 400 / 1500: loss 127.432014
iteration 500 / 1500: loss 86.598565
iteration 600 / 1500: loss 60.329406
iteration 700 / 1500: loss 41.466609
iteration 800 / 1500: loss 29.327829
iteration 900 / 1500: loss 21.278590
iteration 1000 / 1500: loss 16.275040
iteration 1100 / 1500: loss 12.461668
iteration 1200 / 1500: loss 9.803804
iteration 1300 / 1500: loss 9.176964
iteration 1400 / 1500: loss 7.218594
iteration 0 / 1500: loss 788.872211
iteration 100 / 1500: loss 472.226650
iteration 200 / 1500: loss 287.178010
iteration 300 / 1500: loss 175.112846
iteration 400 / 1500: loss 107.845985
iteration 500 / 1500: loss 67.330958
iteration 600 / 1500: loss 42.295300
iteration 700 / 1500: loss 27.952390
iteration 800 / 1500: loss 18.618901
iteration 900 / 1500: loss 13.139217
iteration 1000 / 1500: loss 10.467308
iteration 1100 / 1500: loss 8.271487
iteration 1200 / 1500: loss 7.429124
iteration 1300 / 1500: loss 6.109164
iteration 1400 / 1500: loss 6.288500
iteration 0 / 1500: loss 950.283989
iteration 100 / 1500: loss 515.168160
iteration 200 / 1500: loss 283.149507
iteration 300 / 1500: loss 156.965689
iteration 400 / 1500: loss 88.302851
iteration 500 / 1500: loss 50.056693
iteration 600 / 1500: loss 29.948115
iteration 700 / 1500: loss 19.121930
iteration 800 / 1500: loss 13.103742
iteration 900 / 1500: loss 8.788505
iteration 1000 / 1500: loss 7.654496
iteration 1100 / 1500: loss 6.657006
iteration 1200 / 1500: loss 6.167110
iteration 1300 / 1500: loss 5.514573
iteration 1400 / 1500: loss 5.472784
iteration 0 / 1500: loss 330.470947
iteration 100 / 1500: loss 263.884918
iteration 200 / 1500: loss 216.634511
iteration 300 / 1500: loss 178.301927
iteration 400 / 1500: loss 145.036414
iteration 500 / 1500: loss 118.877483
iteration 600 / 1500: loss 97.335219
iteration 700 / 1500: loss 81.302213
iteration 800 / 1500: loss 66.653045
iteration 900 / 1500: loss 55.942785
iteration 1000 / 1500: loss 46.807460
iteration 1100 / 1500: loss 38.901396
iteration 1200 / 1500: loss 32.564456
iteration 1300 / 1500: loss 27.581567
iteration 1400 / 1500: loss 23.315020
iteration 0 / 1500: loss 635.649783
iteration 100 / 1500: loss 423.904698
iteration 200 / 1500: loss 283.470434
iteration 300 / 1500: loss 191.068545
iteration 400 / 1500: loss 129.042387
iteration 500 / 1500: loss 88.473219
iteration 600 / 1500: loss 60.495922
iteration 700 / 1500: loss 41.847464
iteration 800 / 1500: loss 29.529918
iteration 900 / 1500: loss 21.828547
iteration 1000 / 1500: loss 16.089973
iteration 1100 / 1500: loss 11.956968
iteration 1200 / 1500: loss 9.730971
iteration 1300 / 1500: loss 8.352749
iteration 1400 / 1500: loss 6.989470
iteration 0 / 1500: loss 797.967752
iteration 100 / 1500: loss 476.876654
iteration 200 / 1500: loss 288.993117
iteration 300 / 1500: loss 177.352824
iteration 400 / 1500: loss 108.948804
iteration 500 / 1500: loss 68.252835
iteration 600 / 1500: loss 42.887178
iteration 700 / 1500: loss 28.029709
iteration 800 / 1500: loss 19.100284
iteration 900 / 1500: loss 13.709784
iteration 1000 / 1500: loss 10.585469
iteration 1100 / 1500: loss 8.348889
iteration 1200 / 1500: loss 6.963737
iteration 1300 / 1500: loss 6.101977
iteration 1400 / 1500: loss 5.802818
iteration 0 / 1500: loss 939.010090
iteration 100 / 1500: loss 510.526430
iteration 200 / 1500: loss 281.248266
iteration 300 / 1500: loss 156.053803
iteration 400 / 1500: loss 87.322905
iteration 500 / 1500: loss 49.790121
iteration 600 / 1500: loss 29.508309
iteration 700 / 1500: loss 18.881026
iteration 800 / 1500: loss 12.553106
iteration 900 / 1500: loss 9.373813
iteration 1000 / 1500: loss 7.293139
iteration 1100 / 1500: loss 6.603987
iteration 1200 / 1500: loss 5.876342
iteration 1300 / 1500: loss 5.962211
iteration 1400 / 1500: loss 5.235068
iteration 0 / 1500: loss 332.042130
iteration 100 / 1500: loss 299.082325
iteration 200 / 1500: loss 275.281496
iteration 300 / 1500: loss 252.865674
iteration 400 / 1500: loss 232.742487
iteration 500 / 1500: loss 214.522163
iteration 600 / 1500: loss 197.281979
iteration 700 / 1500: loss 183.024156
iteration 800 / 1500: loss 168.021559
iteration 900 / 1500: loss 156.033116
iteration 1000 / 1500: loss 143.415405
iteration 1100 / 1500: loss 132.747250
iteration 1200 / 1500: loss 122.437723
iteration 1300 / 1500: loss 114.074266
iteration 1400 / 1500: loss 105.929594
iteration 0 / 1500: loss 632.271223
iteration 100 / 1500: loss 536.701240
iteration 200 / 1500: loss 456.130151
iteration 300 / 1500: loss 388.207662
iteration 400 / 1500: loss 330.488396
iteration 500 / 1500: loss 281.833259
iteration 600 / 1500: loss 240.758410
iteration 700 / 1500: loss 206.047084
iteration 800 / 1500: loss 175.245028
iteration 900 / 1500: loss 150.132694
iteration 1000 / 1500: loss 128.717639
iteration 1100 / 1500: loss 109.704455
iteration 1200 / 1500: loss 94.608437
iteration 1300 / 1500: loss 80.777345
iteration 1400 / 1500: loss 69.693331
iteration 0 / 1500: loss 790.026898
iteration 100 / 1500: loss 645.097858
iteration 200 / 1500: loss 527.645901
iteration 300 / 1500: loss 431.876616
iteration 400 / 1500: loss 353.987414
iteration 500 / 1500: loss 291.033988
iteration 600 / 1500: loss 238.067702
iteration 700 / 1500: loss 195.780192
iteration 800 / 1500: loss 160.628948
iteration 900 / 1500: loss 132.558468
iteration 1000 / 1500: loss 108.731622
iteration 1100 / 1500: loss 90.337072
iteration 1200 / 1500: loss 74.510437
iteration 1300 / 1500: loss 61.818274
iteration 1400 / 1500: loss 51.628137
iteration 0 / 1500: loss 937.528987
iteration 100 / 1500: loss 732.351089
iteration 200 / 1500: loss 575.329079
iteration 300 / 1500: loss 454.197334
iteration 400 / 1500: loss 357.704306
iteration 500 / 1500: loss 281.794482
iteration 600 / 1500: loss 222.282219
iteration 700 / 1500: loss 176.224208
iteration 800 / 1500: loss 138.366177
iteration 900 / 1500: loss 110.609362
iteration 1000 / 1500: loss 87.728440
iteration 1100 / 1500: loss 70.135526
iteration 1200 / 1500: loss 55.896838
iteration 1300 / 1500: loss 45.664828
iteration 1400 / 1500: loss 37.035760
lr 2.000000e-08 reg 1.000000e+04 train accuracy: 0.276000 val accuracy: 0.288000
lr 2.000000e-08 reg 2.000000e+04 train accuracy: 0.305000 val accuracy: 0.301000
lr 2.000000e-08 reg 2.500000e+04 train accuracy: 0.323551 val accuracy: 0.324000
lr 2.000000e-08 reg 3.000000e+04 train accuracy: 0.342898 val accuracy: 0.358000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.355980 val accuracy: 0.380000
lr 5.000000e-08 reg 2.000000e+04 train accuracy: 0.377000 val accuracy: 0.396000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.370449 val accuracy: 0.384000
lr 5.000000e-08 reg 3.000000e+04 train accuracy: 0.371347 val accuracy: 0.379000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.383184 val accuracy: 0.385000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.373776 val accuracy: 0.379000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.371857 val accuracy: 0.372000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.360857 val accuracy: 0.375000
best validation accuracy achieved during cross-validation: 0.396000
```

[17]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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

# plot training accuracy
```

```python
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()
```



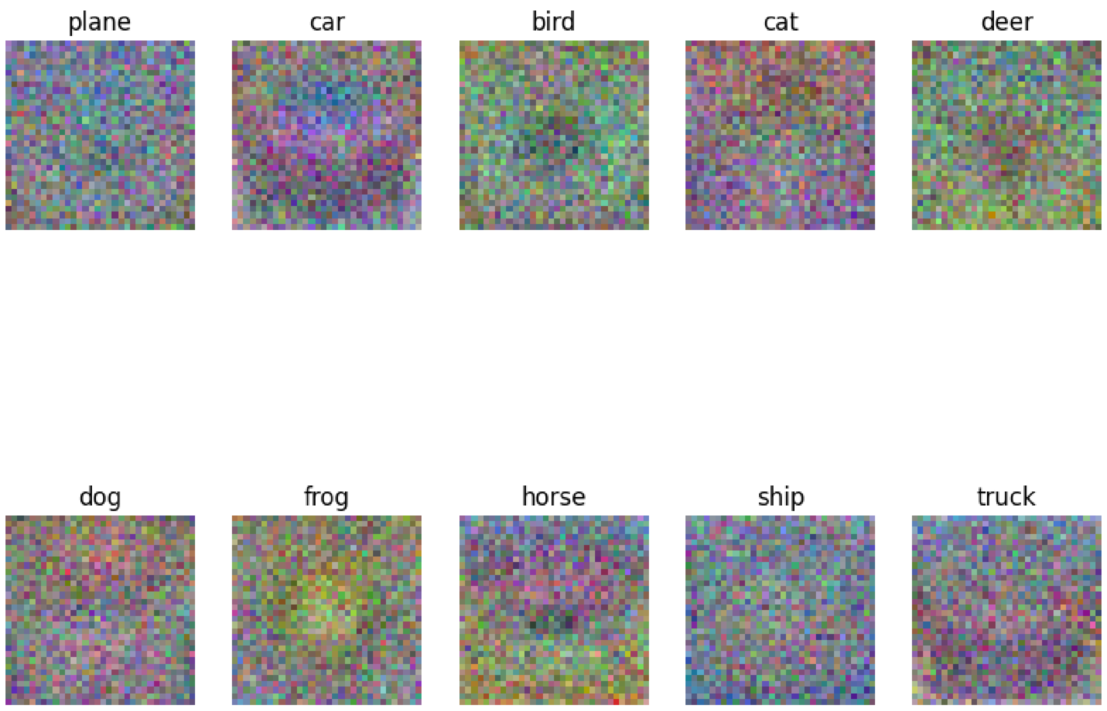


[18]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.377000
```

[19]:
```
# 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 the way they do.

Your Answer : The visualized SVM weights describe the pixel pattern our classifier is using to classify into various classes. For instance, the visualization for frog has a green colored area in the middle - which means that our classifier is predominantly sorting for frog using the green in the middle. Similarly the visualization for deer has green around the outside edge, maybe because most pictures of deer in our training sample were of them in forests or otherwise green surroundings - thus our classifier probably classifies anything in green surroundings to deer (or bird).

softmax

January 22, 2024

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

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cse493g1/assignments/assignment1/'
FOLDERNAME = 'cse493g1/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignments/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment1
```

1 Softmax exercise

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

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

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

[2]:
```
import random
import numpy as np
from cse493g1.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 = 'cse493g1/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 `cse493g1/classifiers/softmax.py`.

[4]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cse493g1/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cse493g1.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: 2.376335
sanity check: 2.302585
```

Inline Question 1

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

$YourAnswer:$ As we initalize our W to very small numbers, we expect to get scores close to 0 for each class in our first iteration. This means e^{s_j} is 1 for all classes, making our softmax loss of $-log\left(\frac{e^{s_j}}{\sum_C e^{s_j}}\right)$ roughly equal to $-log(1/C) = -log(0.1)$ as we have 10 classes.

[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 cse493g1.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

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

```
numerical: 0.010317 analytic: 0.010316, relative error: 5.099403e-06
numerical: -2.305681 analytic: -2.305681, relative error: 2.075644e-08
numerical: -0.421361 analytic: -0.421361, relative error: 1.272277e-07
numerical: -1.576718 analytic: -1.576718, relative error: 3.440082e-08
numerical: 1.568058 analytic: 1.568058, relative error: 4.792425e-08
```

```
numerical: 1.634402 analytic: 1.634402, relative error: 1.862758e-08
numerical: -1.523246 analytic: -1.523246, relative error: 6.413485e-10
numerical: 3.139618 analytic: 3.139618, relative error: 2.624018e-08
numerical: 2.666436 analytic: 2.666435, relative error: 3.062318e-08
numerical: 1.362984 analytic: 1.362984, relative error: 5.431957e-08
numerical: -0.406018 analytic: -0.406018, relative error: 6.022671e-08
numerical: 2.669182 analytic: 2.669182, relative error: 6.598791e-09
numerical: -3.774055 analytic: -3.774055, relative error: 4.276161e-09
numerical: -1.962863 analytic: -1.962863, relative error: 9.124589e-09
numerical: 2.287353 analytic: 2.287353, relative error: 2.379330e-08
numerical: -1.027249 analytic: -1.027249, relative error: 9.084770e-08
numerical: -0.373617 analytic: -0.373617, relative error: 1.495535e-07
numerical: -2.782318 analytic: -2.782318, relative error: 2.266461e-08
numerical: -0.989219 analytic: -0.989220, relative error: 1.383229e-07
numerical: -1.980504 analytic: -1.980504, relative error: 3.618593e-08
```

[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 cse493g1.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: 2.376335e+00 computed in 0.072535s
vectorized loss: 2.376335e+00 computed in 0.010247s
Loss difference: 0.000000
Gradient difference: 0.000000
```

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

from cse493g1.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 = [5e-7, 8e-7]
regularization_strengths = [1e4, 2.5e4]

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

for learn_rate in learning_rates:
  for reg_strength in regularization_strengths:
    softmax = Softmax()
    softmax.train(X_train, y_train, learning_rate= learn_rate,␣
↪reg=reg_strength, num_iters=1500, verbose=True)

    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)

    if val_accuracy > best_val:
      best_val = val_accuracy
      best_softmax = softmax

    results[(learn_rate,reg_strength)] = (train_accuracy, val_accuracy)


# *****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 311.095300
iteration 100 / 1500: loss 42.831225
iteration 200 / 1500: loss 7.412456
iteration 300 / 1500: loss 2.654463
iteration 400 / 1500: loss 2.166783
iteration 500 / 1500: loss 1.987558
iteration 600 / 1500: loss 1.956782
iteration 700 / 1500: loss 2.002824
iteration 800 / 1500: loss 1.937384
iteration 900 / 1500: loss 2.050031
iteration 1000 / 1500: loss 1.932713
iteration 1100 / 1500: loss 2.032762
iteration 1200 / 1500: loss 1.988757
iteration 1300 / 1500: loss 1.893549
iteration 1400 / 1500: loss 1.991416
iteration 0 / 1500: loss 772.070474
iteration 100 / 1500: loss 6.874832
iteration 200 / 1500: loss 2.108995
iteration 300 / 1500: loss 2.056946
iteration 400 / 1500: loss 2.025299
iteration 500 / 1500: loss 2.027340
iteration 600 / 1500: loss 2.091311
iteration 700 / 1500: loss 2.093832
iteration 800 / 1500: loss 2.074885
iteration 900 / 1500: loss 2.101301
iteration 1000 / 1500: loss 2.112404
iteration 1100 / 1500: loss 2.073903
iteration 1200 / 1500: loss 2.181632
iteration 1300 / 1500: loss 2.041569
iteration 1400 / 1500: loss 2.093089
iteration 0 / 1500: loss 312.747297
iteration 100 / 1500: loss 14.090102
iteration 200 / 1500: loss 2.553226
iteration 300 / 1500: loss 2.050413
iteration 400 / 1500: loss 1.987528
iteration 500 / 1500: loss 1.998056
iteration 600 / 1500: loss 2.006035
iteration 700 / 1500: loss 2.060577
iteration 800 / 1500: loss 2.043403
iteration 900 / 1500: loss 2.013145
iteration 1000 / 1500: loss 2.029029
iteration 1100 / 1500: loss 2.018405
iteration 1200 / 1500: loss 1.955473
iteration 1300 / 1500: loss 2.023896
iteration 1400 / 1500: loss 1.984388
iteration 0 / 1500: loss 770.322452
iteration 100 / 1500: loss 2.333117
iteration 200 / 1500: loss 2.024113
iteration 300 / 1500: loss 2.115359
iteration 400 / 1500: loss 2.109861
iteration 500 / 1500: loss 2.143020
iteration 600 / 1500: loss 2.066986
iteration 700 / 1500: loss 2.075460
iteration 800 / 1500: loss 2.081326
iteration 900 / 1500: loss 2.079024
iteration 1000 / 1500: loss 2.146789
iteration 1100 / 1500: loss 2.111555
iteration 1200 / 1500: loss 2.181629
iteration 1300 / 1500: loss 2.106015
iteration 1400 / 1500: loss 2.049183
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.346673 val accuracy: 0.367000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.319041 val accuracy: 0.333000
lr 8.000000e-07 reg 1.000000e+04 train accuracy: 0.344184 val accuracy: 0.355000
lr 8.000000e-07 reg 2.500000e+04 train accuracy: 0.318408 val accuracy: 0.341000
best validation accuracy achieved during cross-validation: 0.367000
```

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

```
softmax on raw pixels final test set accuracy: 0.354000
```

Inline Question 2 - *True or False*

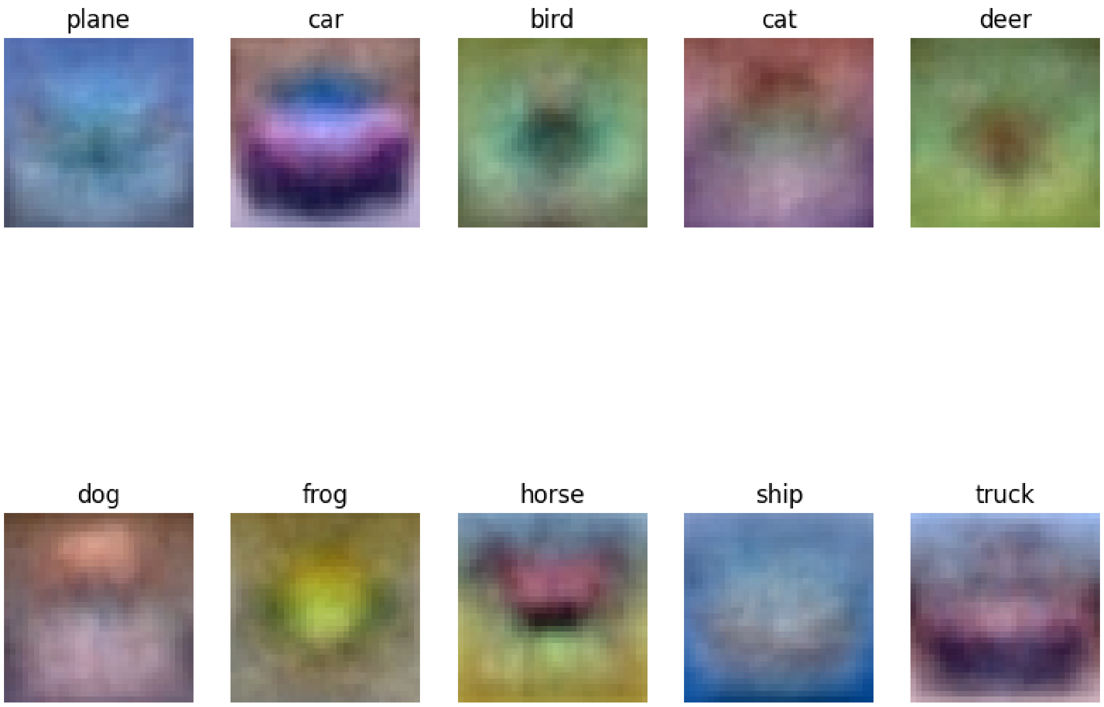
Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$\mathit{YourAnswer:}$ True

$\mathit{YourExplanation:}$ It is possible to have per data point loss $L_i = 0$ for SVM loss if the datapoint is perfectly classified with the declared margin. Adding this point's loss to the overall loss would thus leave it unchanged
However, per data point loss can never be zero for softmax loss. Therefore, adding any point will increase the overall loss

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

```

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

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

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck