knn

April 4, 2023

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.

[33]:
```
from google.colab import drive
drive.mount('/content/drive' , force_remount=True)



#enter your foldername assignments/assignement1
FOLDERNAME = "AA personal/Learning/CS Degree/assignment1/assignment1"



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

import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

#this will download the CIFAR-10 dataset to your drive
#if it isnt already there

%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/AA personal/Learning/CS
Degree/assignment1/assignment1/CV7062610/datasets
--2023-04-04 12:33:28--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

```
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  44.1MB/s    in 3.8s

2023-04-04 12:33:32 (42.7 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

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

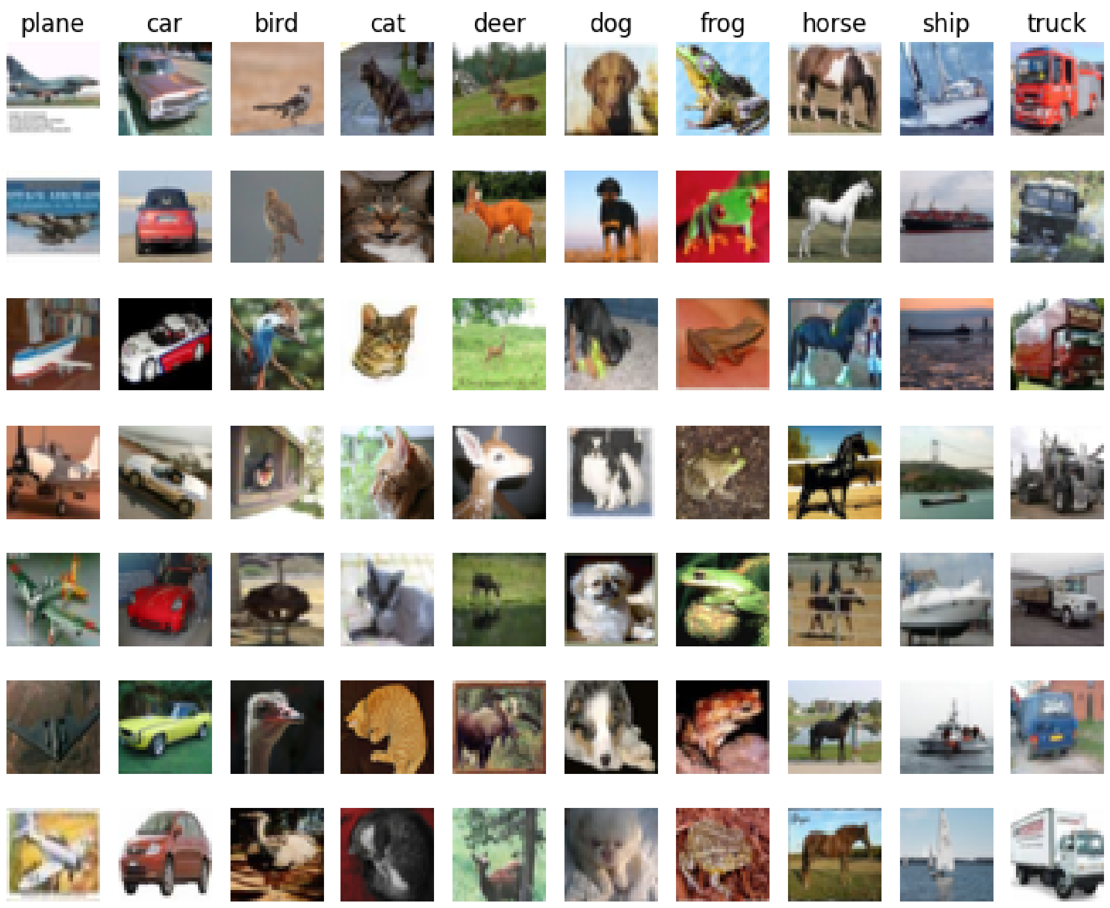
[34]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from CV7062610.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
```

[35]:
```python
# Load the raw CIFAR-10 data.
cifar10_dir = '/content/drive/MyDrive/' + FOLDERNAME + '/CV7062610/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,)
```

[36]:
```python
# 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()
```



[37]:
```
# 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)
```

[38]:
```python
from CV7062610.classifiers import KNearestNeighbor

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

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

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

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

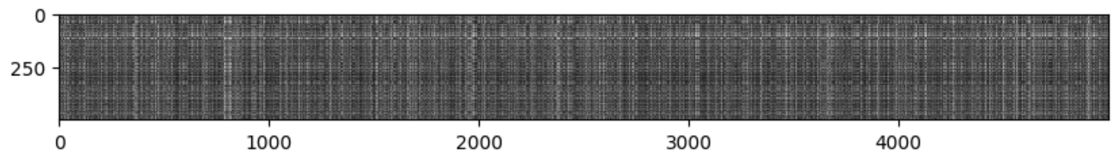
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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

[39]:
```python
# Open CV7062610/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)
```

[40]:
```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 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?

$Your Answer:$ The cause behind the distinctly bright rows are due to that fact that the test instance i at [i,j] is not similar in any maner to any of the training instances in regards to our distance function. a good example would be if we trained our modle on images of cats and dogs (2 classes), then suddenly we test an image of a frong or airaplne. we would rightly so assume that the distance metric would be far off as a frog or airaplne would be totally diffrent than a cat or dog (the data our model was trained on).

what causes the (bright) columns is that an training instance j at [i,j] is not similar in any maner to any of the tested instances in regards to our distance function. a good example would be if we trained our modle on images of cats dogs and cars (3 classes), and our test images were only composed of cats and dogs, hence the column j of a training image of a car would be represented bright white in the dist matrix.

[41]:
```
# 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`:

[42]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

$\color{blue}{Your Answer:}$ (2, 3, 4)

$\color{blue}{Your Explanation:}$ 1 is incorrect as by subtracting a constant (mean over all pixels) would change the images with no relitivaty hence this could change the modles performance. 5 is incorrect as the image has clearly shifted and therefor the knn could change after preprocessing.

2 & 4 is subtracting a relitaive constant pixelwise, hence when the manhattan distance will be computed it will stay relitave pixel wise as each pixel stays relitavly the same in regards to other pixels at that location in the images.

3 is correct although it is not relitave the constant being subtracted it is a sort of normilaztion and there for overall values of pixels will stay relitavely the same in our L1 distance function.

[43]:
```
# 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
```

[49]:
```
# 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
```

[50]:
```
# 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 37.117870 seconds
One loop version took 82.993209 seconds
No loop version took 0.598663 seconds
```

1.0.1 Cross-validation

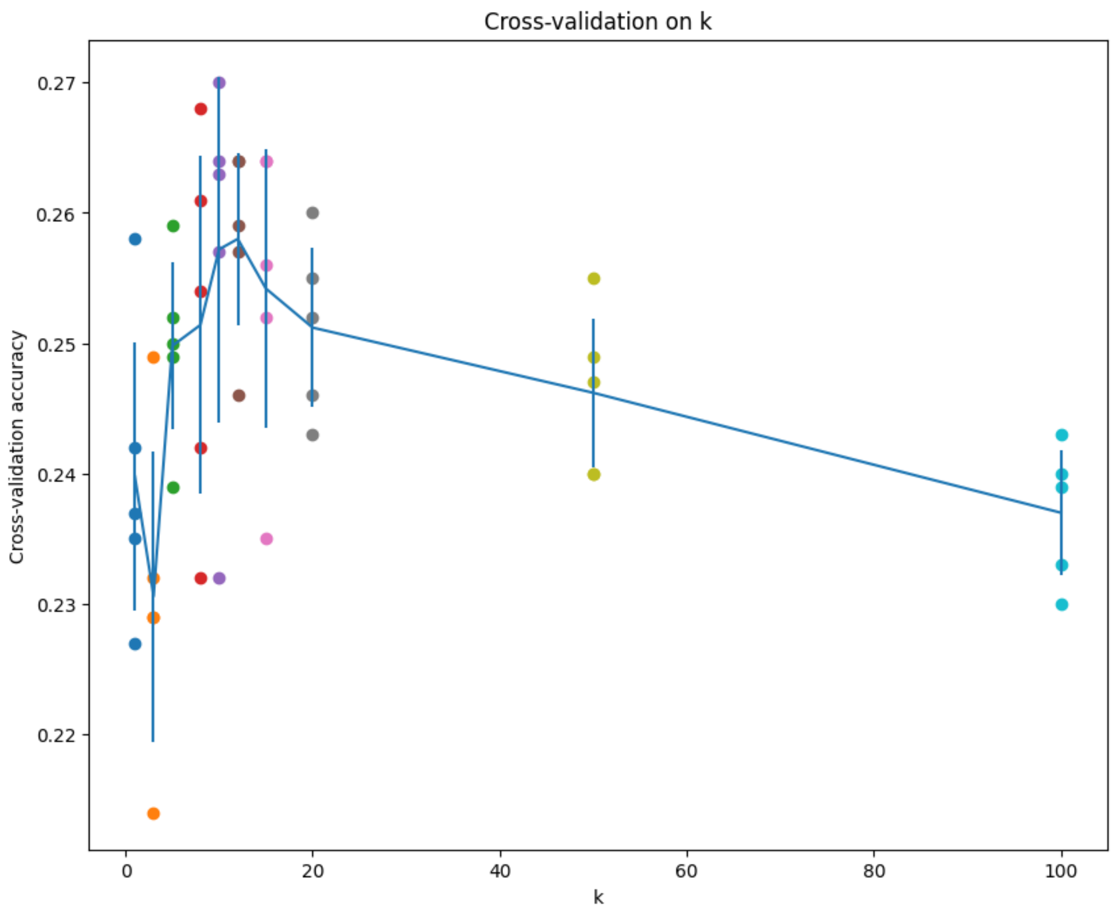
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

[54]:
```
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)*****
# for each k
for k in k_choices:
  #enter dict key for this k
  k_to_accuracies[k] = []
  #make classifier for this k
  classifier = KNearestNeighbor()

  #for takeindex of each numfold
  for t in range(num_folds):
    #for index of each numfold
    for i in range(num_folds):
      #if index not take index add to train
      if i != t:
        X_train_folds_t = np.vstack(X_train_folds[i])
        y_train_folds_t = np.hstack(y_train_folds[i])
        #else take for test
      else:
        X_test_fold = X_train_folds[i]
        y_test_fold = y_train_folds[i]

    #train modle (store the data)
    classifier.train(X_train_folds_t, y_train_folds_t)
    #compute dists matrix
    dists = classifier.compute_distances_no_loops(X_test_fold)
    #prediction of model
    y_test_pred = classifier.predict_labels(dists, k)

    # average over folds as accuracy
    accuracy = np.mean(y_test_pred == y_test_fold)
    #append to dict
    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 = 0.235000
k = 1, accuracy = 0.237000
k = 1, accuracy = 0.258000
k = 1, accuracy = 0.242000
k = 1, accuracy = 0.227000
k = 3, accuracy = 0.229000
k = 3, accuracy = 0.229000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.232000
k = 3, accuracy = 0.214000
k = 5, accuracy = 0.250000
k = 5, accuracy = 0.259000
k = 5, accuracy = 0.252000
k = 5, accuracy = 0.249000
k = 5, accuracy = 0.239000
k = 8, accuracy = 0.254000
k = 8, accuracy = 0.268000
k = 8, accuracy = 0.261000
k = 8, accuracy = 0.242000
k = 8, accuracy = 0.232000
```

```
k = 10, accuracy = 0.263000
k = 10, accuracy = 0.270000
k = 10, accuracy = 0.264000
k = 10, accuracy = 0.257000
k = 10, accuracy = 0.232000
k = 12, accuracy = 0.259000
k = 12, accuracy = 0.264000
k = 12, accuracy = 0.264000
k = 12, accuracy = 0.257000
k = 12, accuracy = 0.246000
k = 15, accuracy = 0.264000
k = 15, accuracy = 0.264000
k = 15, accuracy = 0.256000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.235000
k = 20, accuracy = 0.260000
k = 20, accuracy = 0.252000
k = 20, accuracy = 0.255000
k = 20, accuracy = 0.243000
k = 20, accuracy = 0.246000
k = 50, accuracy = 0.247000
k = 50, accuracy = 0.255000
k = 50, accuracy = 0.240000
k = 50, accuracy = 0.249000
k = 50, accuracy = 0.240000
k = 100, accuracy = 0.243000
k = 100, accuracy = 0.239000
k = 100, accuracy = 0.240000
k = 100, accuracy = 0.230000
k = 100, accuracy = 0.233000
```

[55]:
```
# 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()
```



[56]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

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

$YourAnswer:$ (2, 4)

$YourExplanation:$

answer 5 (none of the above) is clearly incorrect as 4 and 2 are correct.

answer 1 is incorrect as we saw in the lesson the decision boundary is clearly non-liniar (at least in the example we saw , hence not for all k and classification settings).

answer 3 is incorrect. take for example a set of test images where for some reason or other, each image is extremly similar (in regards to the distance function L1 or L2 that we use) to one of the traing images not from its class. so a test image of a frog looks alot like one of the cat images, a test image of a dog looks alot like one of the airoplane images ect... this would resolve in a total disaster and a horrible test error with k=1. however if $k = 5$ in this case we could assume that the other 4 votes would atleast some what improve the test error as in the best case 4 of the remaining votes would vote for the correct class and in some cases at least make the majority of the 5 votes pick the correct class.

answer 4 is correct as we compare each test image to the training set , hence the larger the training set the increase in amount of time shall indeed increase.

answer 2 is correct as the 1-NN training error will be 0 , as it itself is in the traning set hence it shall chose itself with 0 error. (with out loss of genarality in regards to some images having also a 0-error in regards to the image).

softmax

April 4, 2023

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]:
```
from google.colab import drive
drive.mount('/content/drive' , force_remount=True)



#enter your foldername assignments/assignement1

FOLDERNAME = "AA personal/Learning/CS Degree/assignment1/assignment1"

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

import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

#this will download the CIFAR-10 dataset to your drive
#if it isnt already there

%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/AA personal/Learning/CS
Degree/assignment1/assignment1/CV7062610/datasets
--2023-04-04 19:09:31--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

```
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  26.8MB/s    in 6.2s

2023-04-04 19:09:37 (26.3 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

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

[3]:

```
import random
import numpy as np
from CV7062610.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
```

2 New Section

[4]:

```
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 = '/content/drive/MyDrive/' + FOLDERNAME + '/CV7062610/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,)
```

2.1 Softmax Classifier

Your code for this section will all be written inside `CV7062610/classifiers/softmax.py`.

[9]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file CV7062610/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from CV7062610.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: 2.315756
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$

At initialization all scores representing probability's will be roughly equal due to random initialization.

let C be the number of classes in our case C = 10. we would assume the probability of the correct class to be eqaul across the classes due to the random initialization.

as expected we recive,

the pobability of correct class is roughly:

-log(1/C) = log(C) = log(10) = -log(0.1)

[10]:
```
# 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 CV7062610.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: -1.118033 analytic: -1.118033, relative error: 4.430658e-08
numerical: 1.019609 analytic: 1.019609, relative error: 3.137665e-08
numerical: 0.676724 analytic: 0.676724, relative error: 3.650001e-08
numerical: -0.844137 analytic: -0.844137, relative error: 1.072908e-10
numerical: 4.097542 analytic: 4.097542, relative error: 5.528575e-09
numerical: -0.519383 analytic: -0.519383, relative error: 3.685528e-08
numerical: 1.522761 analytic: 1.522761, relative error: 3.127192e-08
numerical: -0.274581 analytic: -0.274581, relative error: 4.246424e-08
numerical: -1.578470 analytic: -1.578470, relative error: 9.964107e-09
numerical: 1.216296 analytic: 1.216296, relative error: 5.303467e-09
numerical: 1.862656 analytic: 1.862655, relative error: 3.783883e-08
numerical: 1.475391 analytic: 1.475391, relative error: 2.138663e-09
numerical: 2.803348 analytic: 2.803348, relative error: 1.224566e-08
numerical: -0.739959 analytic: -0.739959, relative error: 7.946575e-08
numerical: -4.091464 analytic: -4.091464, relative error: 1.546221e-08
numerical: -1.566458 analytic: -1.566458, relative error: 1.282369e-08
numerical: 2.156204 analytic: 2.156204, relative error: 5.043754e-08
numerical: -2.719980 analytic: -2.719980, relative error: 8.784162e-09
```

```
numerical: -0.808292 analytic: -0.808292, relative error: 2.997543e-08
numerical: 0.000399 analytic: 0.000399, relative error: 1.889975e-04
```

[11]:
```
# 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 CV7062610.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: 2.315756e+00 computed in 0.094647s
vectorized loss: 2.315756e+00 computed in 0.012513s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[12]:
```
# 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 CV7062610.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# 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 = [9e-8, 1e-7, 5e-7]
regularization_strengths = [1.8e4, 2e4, 5e4, 2e5]

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

#nested for loop over alpha learning rate and reg strength
for alpha in learning_rates:
    for reg in regularization_strengths:
        #create instance of softMax
        softmax = Softmax()
        #train modle using alpha and reg
        softmax.train(X_train, y_train, learning_rate=alpha, reg=reg,␣
↪num_iters=1500, verbose=True)
        #get predicted values of train and validation
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        #compute val and train accuracy
        accuracy_train = np.mean(y_train == y_train_pred)
        accuracy_val = np.mean(y_val == y_val_pred)
        #store results
        results[(alpha, reg)] = (accuracy_train, accuracy_val)
        #change if beats best
        if accuracy_val > best_val:
            best_val = accuracy_val
            best_softmax = softmax



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

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

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

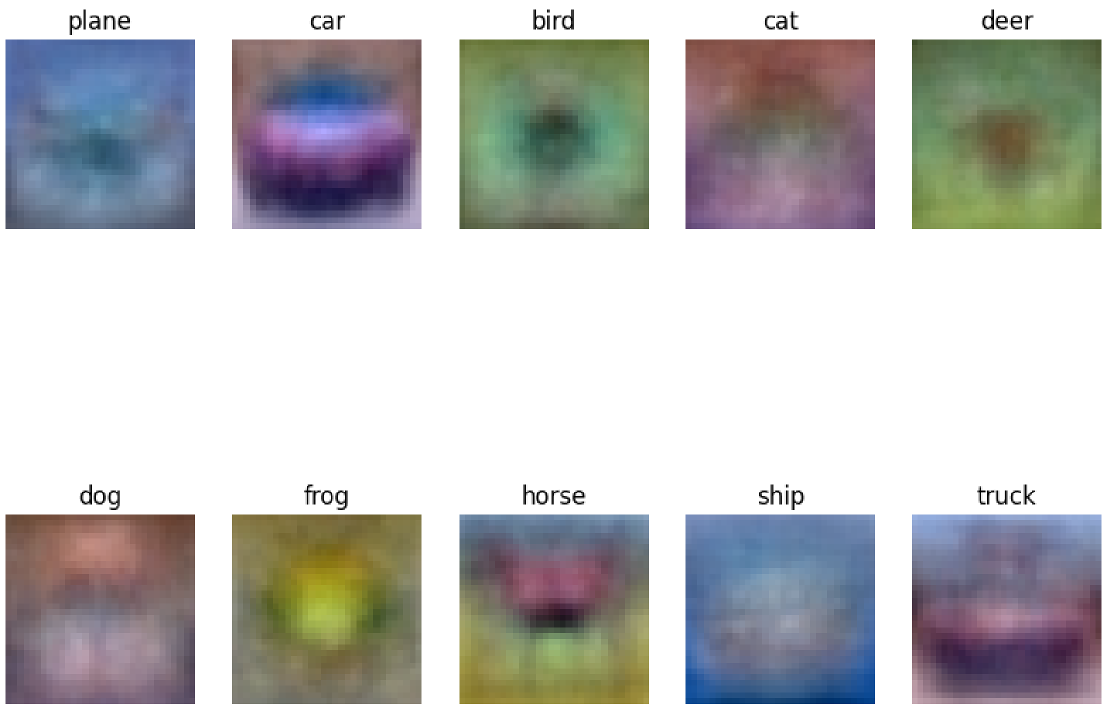
```
iteration 0 / 1500: loss 282.414965
iteration 100 / 1500: loss 203.604794
iteration 200 / 1500: loss 147.137844
iteration 300 / 1500: loss 106.728675
iteration 400 / 1500: loss 77.563370
iteration 500 / 1500: loss 56.599140
iteration 600 / 1500: loss 41.277327
iteration 700 / 1500: loss 30.327311
iteration 800 / 1500: loss 22.490820
iteration 900 / 1500: loss 16.800409
iteration 1000 / 1500: loss 12.665871
iteration 1100 / 1500: loss 9.700309
iteration 1200 / 1500: loss 7.545776
iteration 1300 / 1500: loss 5.950564
iteration 1400 / 1500: loss 4.868958
iteration 0 / 1500: loss 311.690731
iteration 100 / 1500: loss 216.343947
iteration 200 / 1500: loss 150.993479
iteration 300 / 1500: loss 105.718470
iteration 400 / 1500: loss 74.101559
iteration 500 / 1500: loss 52.165902
iteration 600 / 1500: loss 36.984097
iteration 700 / 1500: loss 26.333002
iteration 800 / 1500: loss 18.934670
iteration 900 / 1500: loss 13.846919
iteration 1000 / 1500: loss 10.213029
iteration 1100 / 1500: loss 7.698115
iteration 1200 / 1500: loss 5.966060
iteration 1300 / 1500: loss 4.821684
iteration 1400 / 1500: loss 3.918348
iteration 0 / 1500: loss 792.523271
iteration 100 / 1500: loss 321.013812
iteration 200 / 1500: loss 131.176502
iteration 300 / 1500: loss 54.349677
iteration 400 / 1500: loss 23.231356
iteration 500 / 1500: loss 10.698368
iteration 600 / 1500: loss 5.578004
iteration 700 / 1500: loss 3.537530
iteration 800 / 1500: loss 2.591268
iteration 900 / 1500: loss 2.330727
iteration 1000 / 1500: loss 2.191909
iteration 1100 / 1500: loss 2.144483
iteration 1200 / 1500: loss 2.142232
iteration 1300 / 1500: loss 2.080929
iteration 1400 / 1500: loss 2.110679
iteration 0 / 1500: loss 3033.283053
iteration 100 / 1500: loss 82.086270
iteration 200 / 1500: loss 4.299370
iteration 300 / 1500: loss 2.226338
iteration 400 / 1500: loss 2.216366
iteration 500 / 1500: loss 2.216793
iteration 600 / 1500: loss 2.205293
iteration 700 / 1500: loss 2.167035
iteration 800 / 1500: loss 2.201938
iteration 900 / 1500: loss 2.206941
iteration 1000 / 1500: loss 2.190012
iteration 1100 / 1500: loss 2.168763
iteration 1200 / 1500: loss 2.156823
iteration 1300 / 1500: loss 2.163299
iteration 1400 / 1500: loss 2.188753
iteration 0 / 1500: loss 282.646742
iteration 100 / 1500: loss 196.298454
iteration 200 / 1500: loss 137.177673
iteration 300 / 1500: loss 95.906387
iteration 400 / 1500: loss 67.334880
iteration 500 / 1500: loss 47.430027
iteration 600 / 1500: loss 33.637878
iteration 700 / 1500: loss 24.060690
iteration 800 / 1500: loss 17.356693
iteration 900 / 1500: loss 12.691442
iteration 1000 / 1500: loss 9.516906
iteration 1100 / 1500: loss 7.121526
iteration 1200 / 1500: loss 5.604951
iteration 1300 / 1500: loss 4.520763
iteration 1400 / 1500: loss 3.763688
iteration 0 / 1500: loss 309.076413
iteration 100 / 1500: loss 206.299111
iteration 200 / 1500: loss 138.319842
iteration 300 / 1500: loss 93.015273
iteration 400 / 1500: loss 62.956065
iteration 500 / 1500: loss 42.774098
iteration 600 / 1500: loss 29.322823
iteration 700 / 1500: loss 20.207290
iteration 800 / 1500: loss 14.175203
iteration 900 / 1500: loss 10.113983
iteration 1000 / 1500: loss 7.449579
iteration 1100 / 1500: loss 5.660832
iteration 1200 / 1500: loss 4.475981
iteration 1300 / 1500: loss 3.701398
iteration 1400 / 1500: loss 3.114634
iteration 0 / 1500: loss 776.369642
iteration 100 / 1500: loss 284.587238
iteration 200 / 1500: loss 105.439814
iteration 300 / 1500: loss 39.881067
iteration 400 / 1500: loss 15.877091
iteration 500 / 1500: loss 7.152467
iteration 600 / 1500: loss 3.963735
iteration 700 / 1500: loss 2.757922
iteration 800 / 1500: loss 2.325259
iteration 900 / 1500: loss 2.126089
iteration 1000 / 1500: loss 2.120036
iteration 1100 / 1500: loss 2.082713
iteration 1200 / 1500: loss 2.082984
iteration 1300 / 1500: loss 2.080655
iteration 1400 / 1500: loss 2.074440
iteration 0 / 1500: loss 3082.049152
iteration 100 / 1500: loss 56.090286
iteration 200 / 1500: loss 3.125234
iteration 300 / 1500: loss 2.220968
iteration 400 / 1500: loss 2.204446
iteration 500 / 1500: loss 2.203462
iteration 600 / 1500: loss 2.180036
iteration 700 / 1500: loss 2.211669
iteration 800 / 1500: loss 2.185515
iteration 900 / 1500: loss 2.171939
iteration 1000 / 1500: loss 2.219227
iteration 1100 / 1500: loss 2.183594
iteration 1200 / 1500: loss 2.214083
iteration 1300 / 1500: loss 2.137427
iteration 1400 / 1500: loss 2.178025
iteration 0 / 1500: loss 283.941169
iteration 100 / 1500: loss 47.267638
iteration 200 / 1500: loss 9.360768
iteration 300 / 1500: loss 3.253865
iteration 400 / 1500: loss 2.223923
iteration 500 / 1500: loss 2.013226
iteration 600 / 1500: loss 2.019998
iteration 700 / 1500: loss 1.951721
iteration 800 / 1500: loss 1.896052
iteration 900 / 1500: loss 2.043699
iteration 1000 / 1500: loss 1.876892
iteration 1100 / 1500: loss 2.089189
iteration 1200 / 1500: loss 1.949197
iteration 1300 / 1500: loss 2.019904
iteration 1400 / 1500: loss 2.022000
iteration 0 / 1500: loss 312.357031
iteration 100 / 1500: loss 42.927999
iteration 200 / 1500: loss 7.449401
iteration 300 / 1500: loss 2.672904
iteration 400 / 1500: loss 2.060357
iteration 500 / 1500: loss 1.971345
iteration 600 / 1500: loss 2.081279
iteration 700 / 1500: loss 1.981028
iteration 800 / 1500: loss 2.011267
iteration 900 / 1500: loss 2.059740
iteration 1000 / 1500: loss 2.014995
iteration 1100 / 1500: loss 1.976109
iteration 1200 / 1500: loss 1.946866
iteration 1300 / 1500: loss 1.965733
iteration 1400 / 1500: loss 2.054041
iteration 0 / 1500: loss 765.195113
iteration 100 / 1500: loss 6.796174
iteration 200 / 1500: loss 2.132396
iteration 300 / 1500: loss 2.137279
iteration 400 / 1500: loss 2.059826
iteration 500 / 1500: loss 2.116592
iteration 600 / 1500: loss 2.090112
iteration 700 / 1500: loss 2.103206
iteration 800 / 1500: loss 2.089820
iteration 900 / 1500: loss 2.137460
iteration 1000 / 1500: loss 2.052101
iteration 1100 / 1500: loss 2.113070
iteration 1200 / 1500: loss 2.115139
iteration 1300 / 1500: loss 2.064337
iteration 1400 / 1500: loss 2.073730
iteration 0 / 1500: loss 3062.756437
iteration 100 / 1500: loss 2.200957
iteration 200 / 1500: loss 2.176494
iteration 300 / 1500: loss 2.191929
iteration 400 / 1500: loss 2.211855
iteration 500 / 1500: loss 2.151666
iteration 600 / 1500: loss 2.196710
iteration 700 / 1500: loss 2.188884
iteration 800 / 1500: loss 2.198145
iteration 900 / 1500: loss 2.184595
iteration 1000 / 1500: loss 2.224089
iteration 1100 / 1500: loss 2.226018
iteration 1200 / 1500: loss 2.182660
iteration 1300 / 1500: loss 2.197443
iteration 1400 / 1500: loss 2.166466
lr 9.000000e-08 reg 1.800000e+04 train accuracy: 0.348878 val accuracy: 0.352000
lr 9.000000e-08 reg 2.000000e+04 train accuracy: 0.351510 val accuracy: 0.362000
lr 9.000000e-08 reg 5.000000e+04 train accuracy: 0.329551 val accuracy: 0.344000
lr 9.000000e-08 reg 2.000000e+05 train accuracy: 0.286490 val accuracy: 0.302000
lr 1.000000e-07 reg 1.800000e+04 train accuracy: 0.354163 val accuracy: 0.372000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.354755 val accuracy: 0.368000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.331163 val accuracy: 0.343000
lr 1.000000e-07 reg 2.000000e+05 train accuracy: 0.291571 val accuracy: 0.303000
lr 5.000000e-07 reg 1.800000e+04 train accuracy: 0.356224 val accuracy: 0.377000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.353102 val accuracy: 0.361000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.324204 val accuracy: 0.335000
lr 5.000000e-07 reg 2.000000e+05 train accuracy: 0.289061 val accuracy: 0.298000
best validation accuracy achieved during cross-validation: 0.377000
```

[13]:
```
# 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: 0.356000
```

[14]:
```
# 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

April 4, 2023

1 Implementing a Neural Network

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

[5]:
```
import numpy as np
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive' , force_remount=True)

FOLDERNAME = "AA personal/Learning/CS Degree/assignment1/assignment1"

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

import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

from CV7062610.classifiers.neural_net import TwoLayerNet

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

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

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

```
Mounted at /content/drive
```

We will use the class `TwoLayerNet` in the file `CV7062610/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

[6]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

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

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

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

2 Forward pass: compute scores

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

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

[42]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()
```

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

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

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

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

3 Forward pass: compute loss

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

[48]:
```python
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

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

```
Difference between your loss and correct loss:
1.6321838670026707
```

4 Backward pass

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

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

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

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

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

5 Train the network

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

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

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

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

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

6 Load the data

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

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

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

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

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

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

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

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

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


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

7 Train a network

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

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

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

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

8 Debug the training

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

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

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

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

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

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

# Visualize the weights of the network

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

show_net_weights(net)
```

9 Tune your hyperparameters

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

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

Explain your hyperparameter tuning process below.

$Your Answer :$

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

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

     pass

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

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

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

10 Run on the test set

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

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

Inline Question

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

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

$Your Answer:$

$Your Explanation:$