knn

April 23, 2020

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

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

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

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

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

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

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-23 04:06:16--  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  74.5MB/s    in 2.2s

2020-04-23 04:06:18 (74.5 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

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

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

[0]:
```
# Run some setup code for this notebook.

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

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

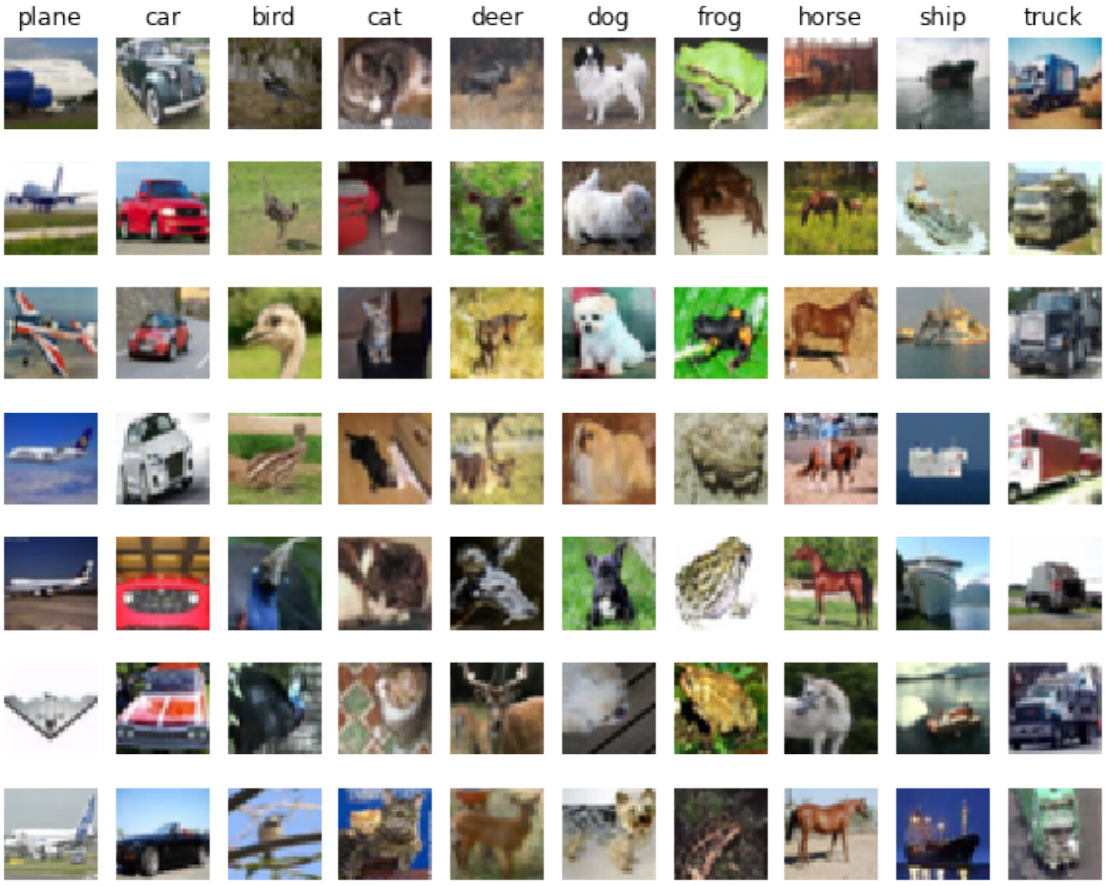
```
# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[0]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

[0]:
```
# 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()
```



[0]:
```
# 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)
```

[0]:
```
from cs231n.classifiers import KNearestNeighbor

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

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

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

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

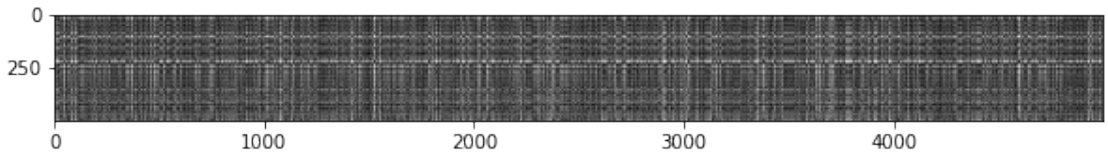
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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

[0]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

[0]:
```
# 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 :

We are comparing the distance between training images and test images with pixel values. bright color indicate high distances between two images, the pixel values between two images have a large difference. The ith column represents the ith training image and the jth row represents the jth test image. (i,j) means comparing ith training image to jth test image.

[0]:
```
# 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`:

[0]:
```
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.

YourAnswer :

1 & 2 & 3

YourExplanation : \ 1. The distance will be the same if we subtract mean for each pixel. \ 2. The distance will be the same if we subtract pixel mean for each pixel. \ 3. The order of distance will be the same if subtracting mean and dividing by the standard deviation. \ 4. If dividing by pixel-wise standard deviation, the distance between each image will be different. \ 5. If rotating a matrix by a certain degree, the distance between each data will be different.

[0]:
```
# 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
```

[0]:
```
# 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
```

[0]:
```
# 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 36.283157 seconds
One loop version took 32.024806 seconds
No loop version took 0.540124 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.

[0]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        ␣
 ↪#
# Split up the training data into folds. After splitting, X_train_folds and   ␣
 ↪#
```

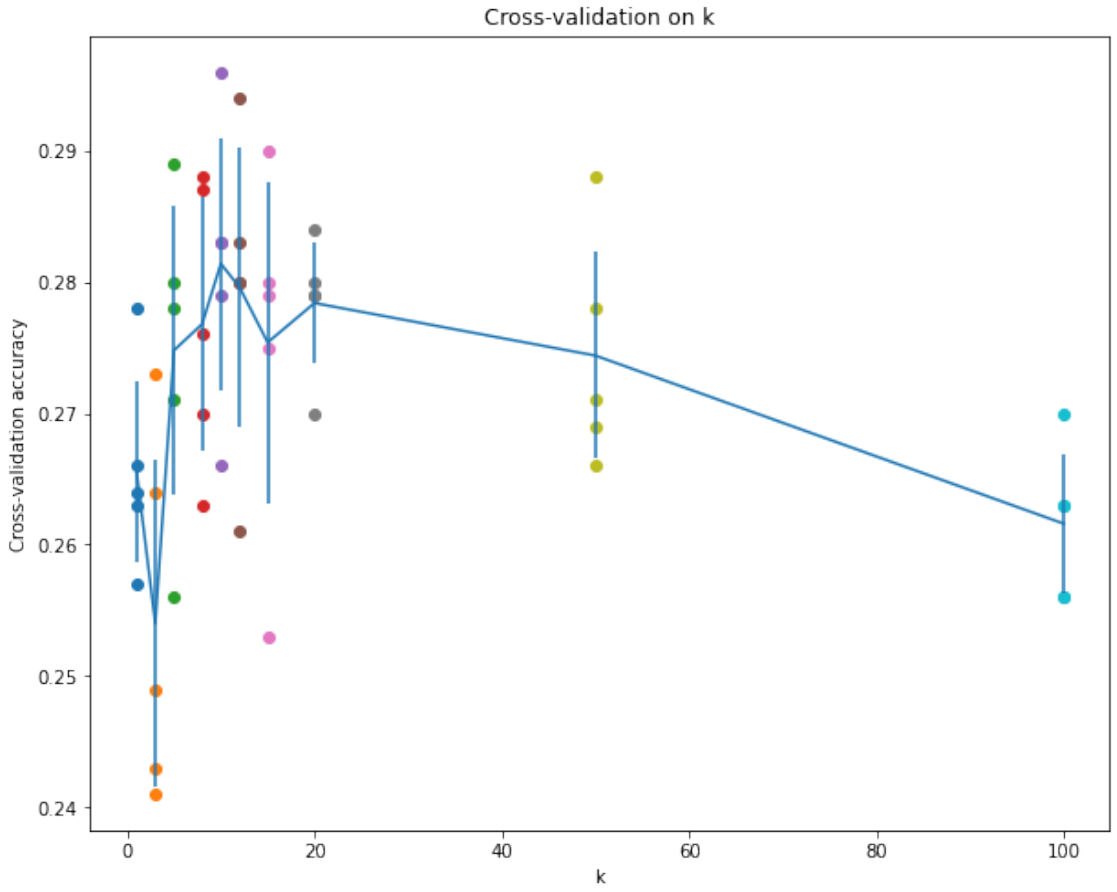
```
# y_train_folds should each be lists of length num_folds, where
# y_train_folds[i] is the label vector for the points in X_train_folds[i].
# Hint: Look up the numpy array_split function.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

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

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

for k in k_choices:
  validation_accuracies = []
  for i in range(num_folds):
    x_trn = np.concatenate((x_train_folds[:i]+x_train_folds[i+1:]))
    y_trn = np.concatenate((y_train_folds[:i]+y_train_folds[i+1:]))
    classifier.train(x_trn,y_trn)
    Yval_predict = classifier.predict(x_train_folds[i], k = k)
    num_correct = np.sum(Yval_predict == y_train_folds[i])
    accuracy = float(num_correct) / len(x_train_folds[0])
    validation_accuracies.append(accuracy)
```

```
  k_to_accuracies[k] = validation_accuracies


# *****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 = 0.241000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.243000
k = 3, accuracy = 0.273000
k = 3, accuracy = 0.264000
k = 5, accuracy = 0.256000
k = 5, accuracy = 0.271000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.289000
k = 5, accuracy = 0.278000
k = 8, accuracy = 0.263000
k = 8, accuracy = 0.287000
k = 8, accuracy = 0.276000
k = 8, accuracy = 0.288000
k = 8, accuracy = 0.270000
k = 10, accuracy = 0.266000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.279000
k = 10, accuracy = 0.283000
k = 10, accuracy = 0.283000
k = 12, accuracy = 0.261000
k = 12, accuracy = 0.294000
k = 12, accuracy = 0.280000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.253000
k = 15, accuracy = 0.290000
k = 15, accuracy = 0.279000
k = 15, accuracy = 0.280000
k = 15, accuracy = 0.275000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
```

```
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.280000
k = 20, accuracy = 0.284000
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
```

[0]:
```
# 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()
```



[0]:
```
# 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 :

1. No. k-NN is not a linear classifier.
2. Yes. If given a point, the nearest neighbor will be the exact same point. Thus, the error will be zero.
3. No. If a test data point is not the same as any training data point, 1-NN will not be better than 5-NN.
4. Yes. K-NN will need to loop through all the traning example and compute the distance of each test-train set and sort the distance.

---

2 IMPORTANT

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

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

[0]:
```
import os

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

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

[0]:
```

```

svm

April 23, 2020

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

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

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

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

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

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

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-23 04:09:05--  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  46.8MB/s    in 3.8s

2020-04-23 04:09:09 (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
```

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

[0]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

# Some more magic so that the notebook will reload external python modules;
```

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

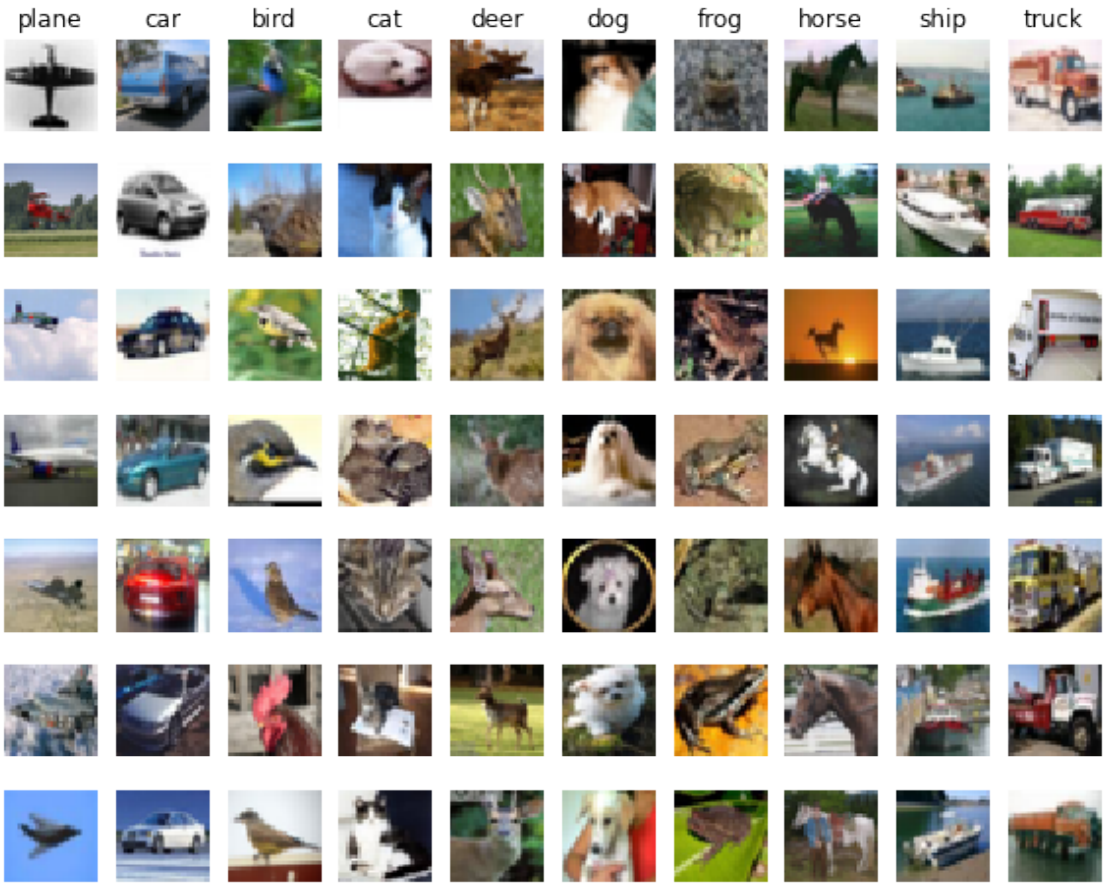
1.1 CIFAR-10 Data Loading and Preprocessing

[0]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

[0]:
```
# 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()
```



[0]:
```
# 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)
```

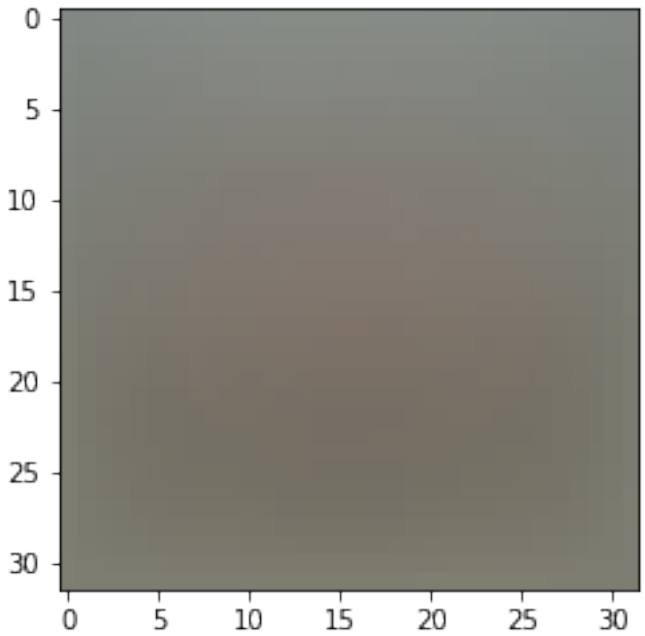
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[0]:
```
# 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)
```

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

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

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

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

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



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

1.2 SVM Classifier

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

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

[0]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 8.957905
```

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

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

[0]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

```
numerical: 1.243886 analytic: 1.243886, relative error: 2.099809e-10
numerical: -9.106280 analytic: -9.106280, relative error: 1.153445e-12
```

```
numerical: 11.668000 analytic: 11.668000, relative error: 2.293903e-12
numerical: -30.776785 analytic: -30.776785, relative error: 6.078395e-12
numerical: -4.599057 analytic: -4.599057, relative error: 2.831601e-11
numerical: -13.083475 analytic: -13.083475, relative error: 1.185621e-12
numerical: -5.762000 analytic: -5.762000, relative error: 4.414672e-11
numerical: 22.027540 analytic: 22.027540, relative error: 2.577013e-12
numerical: 12.413586 analytic: 12.413586, relative error: 1.083186e-11
numerical: 4.777295 analytic: 4.777295, relative error: 1.090215e-12
numerical: 4.103215 analytic: 4.103215, relative error: 9.523975e-11
numerical: 8.041344 analytic: 8.041344, relative error: 1.061461e-11
numerical: 11.546394 analytic: 11.546394, relative error: 5.746194e-12
numerical: -48.217441 analytic: -48.217441, relative error: 1.294223e-11
numerical: 13.043819 analytic: 13.043819, relative error: 1.743738e-11
numerical: -26.020208 analytic: -26.020208, relative error: 3.916009e-12
numerical: -5.746525 analytic: -5.746525, relative error: 4.432109e-11
numerical: -41.507490 analytic: -41.507490, relative error: 5.419350e-12
numerical: 3.829303 analytic: 3.829303, relative error: 5.229041e-11
numerical: -6.373626 analytic: -6.373626, relative error: 1.160894e-11
```

Inline Question 1

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

Your Answer :

The SVM loss is max(0,s), s will be the score difference between correct and incorrect classes plus a constant. This will make SVM undifferentiable when at zero.

[0]:
```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
Naive loss: 8.957905e+00 computed in 0.116003s
Vectorized loss: 8.957905e+00 computed in 0.012680s
difference: -0.000000
```

[0]:
```
# 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 0.117858s
Vectorized loss and gradient: computed in 0.009590s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

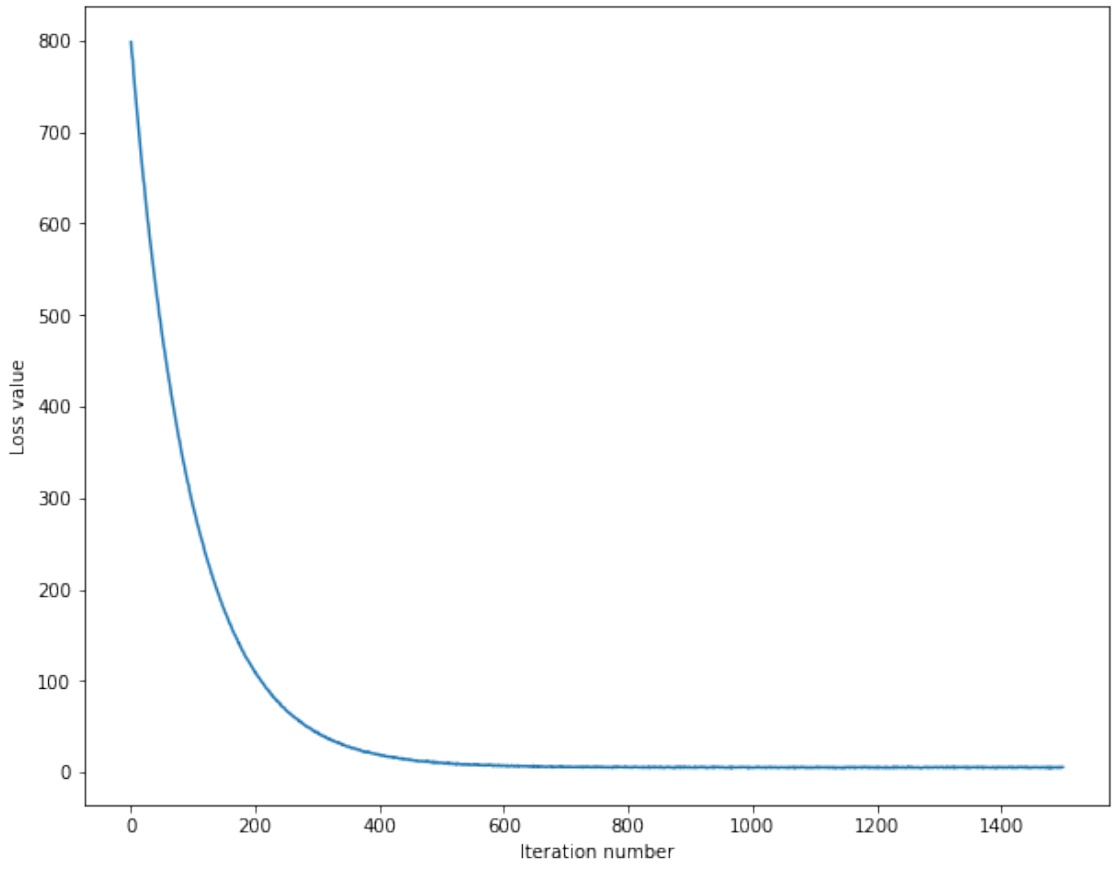
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[0]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 798.426415
iteration 100 / 1500: loss 290.681046
iteration 200 / 1500: loss 109.597516
iteration 300 / 1500: loss 43.136711
iteration 400 / 1500: loss 18.991680
iteration 500 / 1500: loss 10.428846
iteration 600 / 1500: loss 7.290796
```

```
iteration 700 / 1500: loss 6.020245
iteration 800 / 1500: loss 5.444534
iteration 900 / 1500: loss 5.290623
iteration 1000 / 1500: loss 5.391263
iteration 1100 / 1500: loss 5.417947
iteration 1200 / 1500: loss 5.287551
iteration 1300 / 1500: loss 5.172803
iteration 1400 / 1500: loss 5.504787
That took 8.887400s
```

[0]:
```
# 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()
```



[0]:
```
# 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: 0.364184
validation accuracy: 0.384000
```

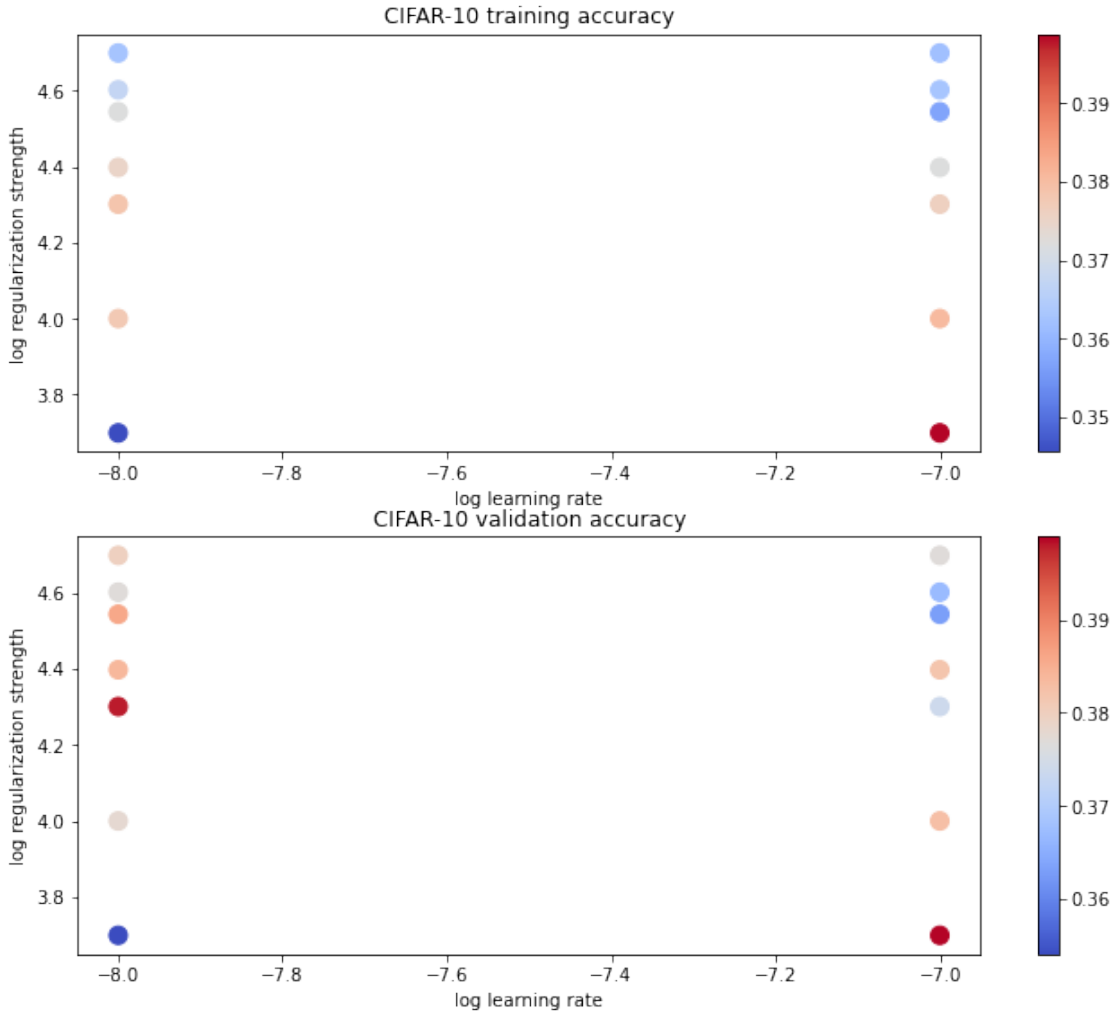
[0]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

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

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

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

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

# Provided as a reference. You may or may not want to change these
hyperparameters
learning_rates = [1e-7,1e-8]
regularization_strengths = [5e3,1e3,1e4,2e4,2.5e4,3.5e4,4e4,5e4]
# lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.398857 val accuracy: 0.
399000
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for learning_rate in learning_rates:
  for regularization_strength in regularization_strengths:
    svm = LinearSVM()
    loss_hist = svm.train(X_train, y_train, learning_rate,
regularization_strength, num_iters=10000, verbose=False)
    y_train_pred = svm.predict(X_train)
    train_acc = np.mean(y_train_pred==y_train)
    y_val_pred = svm.predict(X_val)
    val_acc = np.mean(y_val_pred==y_val)
    if best_val < val_acc:
      best_val = val_acc
      best_svm = svm
    results[(learning_rate, regularization_strength)] = (train_acc,val_acc)
# *****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)
```

```
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.345510 val accuracy: 0.354000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.377959 val accuracy: 0.378000
lr 1.000000e-08 reg 2.000000e+04 train accuracy: 0.378673 val accuracy: 0.398000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.375245 val accuracy: 0.384000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.372061 val accuracy: 0.386000
lr 1.000000e-08 reg 4.000000e+04 train accuracy: 0.367429 val accuracy: 0.377000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.362612 val accuracy: 0.380000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.398857 val accuracy: 0.399000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.380633 val accuracy: 0.383000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.376265 val accuracy: 0.374000
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.372000 val accuracy: 0.382000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.357102 val accuracy: 0.363000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.362898 val accuracy: 0.367000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.361918 val accuracy: 0.377000
best validation accuracy achieved during cross-validation: 0.399000
```

[0]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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

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

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

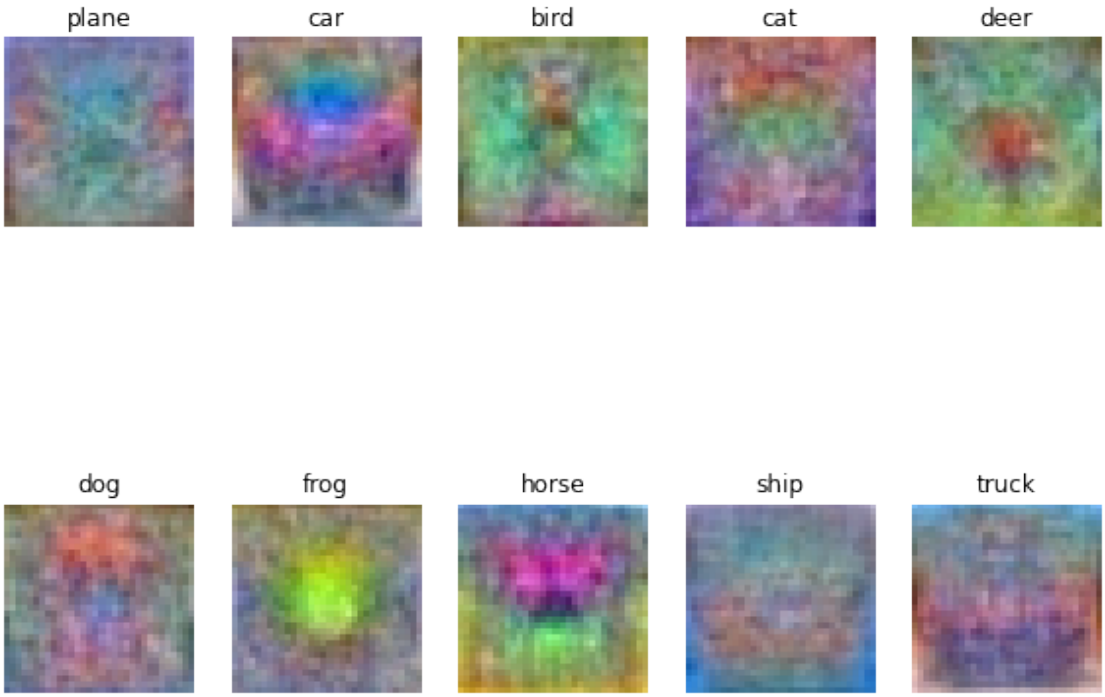


[0]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.380000
```

[0]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
 ↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
```

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

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



Inline question 2

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

Your Answer :

The results look blurry and look like a combination of all different images. It might because SVM use W to calculate scores of each image. The way SVM visualizes the images is only using 1 weight matrix, W, to calculate the scores and distinguish from different classes.

---

2 IMPORTANT

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

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

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

[0]:
```
import os

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

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

[0]:
```

```

softmax

April 23, 2020

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

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

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

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

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

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

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-23 04:22:34--  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  47.7MB/s    in 3.6s

2020-04-23 04:22:38 (45.6 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

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

1 Softmax exercise

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

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

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

[0]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

```
%autoreload 2
```

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

[0]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

    # Normalize the data: subtract the mean image
```

```
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

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

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


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

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

1.1 Softmax Classifier

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

[0]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
```

```
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

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

```
loss: 2.386125
sanity check: 2.302585
```

Inline Question 1

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

Your Answer :

Since softmax loss is -log(probablity of correct class), therefore, initially, the probablity to select to the correct class will be 0.1 for 10 different classes data set.

[0]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

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

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

```
numerical: -0.395897 analytic: -0.395897, relative error: 1.704795e-08
numerical: -0.060262 analytic: -0.060262, relative error: 2.410020e-07
numerical: -0.308514 analytic: -0.308514, relative error: 2.702203e-07
numerical: 0.408854 analytic: 0.408854, relative error: 1.771834e-08
numerical: 1.068561 analytic: 1.068561, relative error: 5.465436e-08
numerical: 0.238180 analytic: 0.238180, relative error: 2.617836e-07
numerical: -3.214318 analytic: -3.214318, relative error: 3.404385e-09
numerical: -0.970356 analytic: -0.970356, relative error: 4.621204e-09
numerical: 0.168110 analytic: 0.168110, relative error: 2.242742e-07
numerical: 3.507259 analytic: 3.507259, relative error: 1.908073e-08
numerical: 1.333792 analytic: 1.333792, relative error: 7.065376e-09
numerical: 1.801662 analytic: 1.801662, relative error: 3.160302e-08
numerical: 0.139000 analytic: 0.139000, relative error: 2.568892e-07
numerical: -1.008915 analytic: -1.008915, relative error: 4.835153e-08
numerical: 0.244451 analytic: 0.244451, relative error: 1.001400e-07
numerical: 1.351789 analytic: 1.351789, relative error: 4.606441e-08
numerical: 1.423210 analytic: 1.423210, relative error: 2.428566e-08
```

```
numerical: -0.867374 analytic: -0.867374, relative error: 2.835960e-08
numerical: 1.865261 analytic: 1.865261, relative error: 2.021049e-08
numerical: 1.479380 analytic: 1.479379, relative error: 7.053801e-08
```

[0]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
naive loss: 2.386125e+00 computed in 0.126101s
vectorized loss: 2.386125e+00 computed in 0.010242s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[0]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

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

################################################################################
# TODO:                                                                        #
```

```
# Use the validation set to set the learning rate and regularization strength.
#
# This should be identical to the validation that you did for the SVM; save
#
# the best trained softmax classifer in best_softmax.
#
################################################################################

# Provided as a reference. You may or may not want to change these
hyperparameters
learning_rates = [5e-7,1e-7,1e-8]
regularization_strengths = [5e3,1e3,1e4,2e4,2.5e4,3.5e4,4e4,5e4]
#lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.403327 val accuracy: 0.
407000
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for learning_rate in learning_rates:
  for regularization_strength in regularization_strengths:
    softmax = Softmax()
    loss_hist = softmax.train(X_train, y_train, learning_rate,
regularization_strength, num_iters=5000, verbose=False)
    y_train_pred = softmax.predict(X_train)
    train_acc = np.mean(y_train_pred==y_train)
    y_val_pred = softmax.predict(X_val)
    val_acc = np.mean(y_val_pred==y_val)
    if best_val < val_acc:
      best_val = val_acc
      best_softmax = softmax
    results[(learning_rate, regularization_strength)] = (train_acc,val_acc)
# *****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)
```

```
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.203306 val accuracy: 0.212000
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.226816 val accuracy: 0.246000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.253673 val accuracy: 0.264000
lr 1.000000e-08 reg 2.000000e+04 train accuracy: 0.304449 val accuracy: 0.303000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.306918 val accuracy: 0.328000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.314816 val accuracy: 0.337000
lr 1.000000e-08 reg 4.000000e+04 train accuracy: 0.310653 val accuracy: 0.319000
```

```
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.306878 val accuracy: 0.330000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.363286 val accuracy: 0.387000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.374082 val accuracy: 0.385000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.360102 val accuracy: 0.377000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.339327 val accuracy: 0.350000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.327286 val accuracy: 0.351000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.323306 val accuracy: 0.337000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.313633 val accuracy: 0.329000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.309041 val accuracy: 0.326000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.403327 val accuracy: 0.407000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.372469 val accuracy: 0.377000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.353796 val accuracy: 0.359000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.329653 val accuracy: 0.343000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.329224 val accuracy: 0.343000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.297551 val accuracy: 0.308000
lr 5.000000e-07 reg 4.000000e+04 train accuracy: 0.320857 val accuracy: 0.339000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.303102 val accuracy: 0.318000
best validation accuracy achieved during cross-validation: 0.407000
```

[0]:
```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: 0.384000
```

Inline Question 2 - *True or False*

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

YourAnswer :

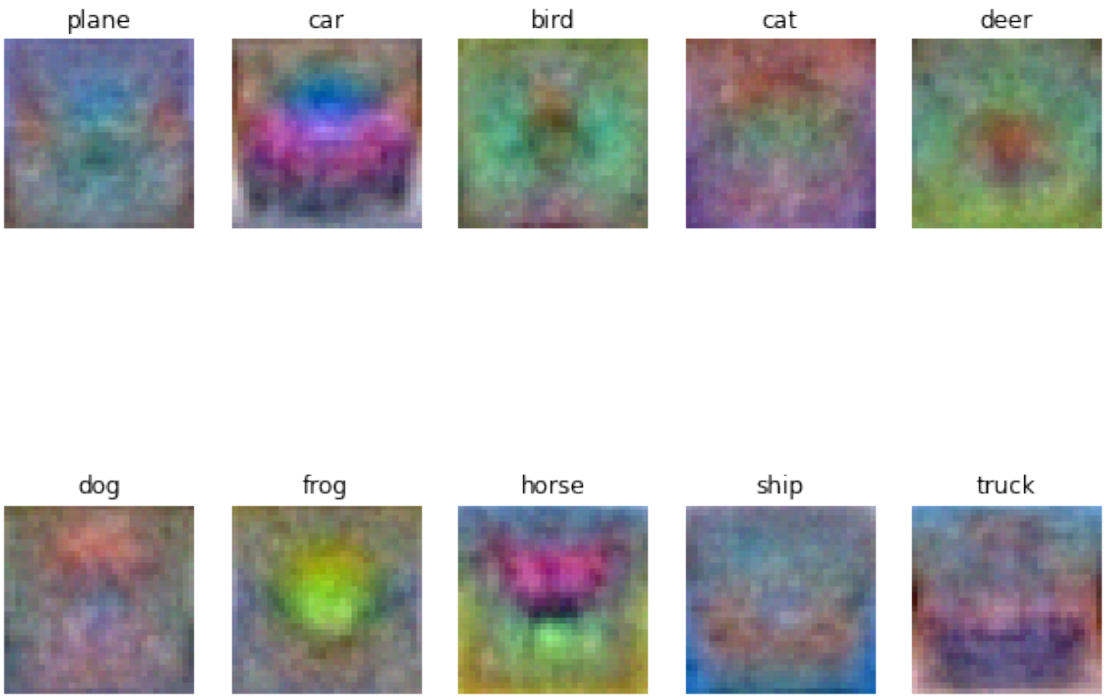
True

YourExplanation :

Consider an example that achieves the scores [10,-2,3] and where the first class is correct. The SVM does not care about the details of the individual scores: if they were instead [10,-100,-100] or [10,9,9] the SVM would be indifferent since the margin of 1 is satified and hence the loss is zero. However, these scenarios are not equivalent to a softmax classifier where the loss will increase and would accumulate a much higher loss for the scores [10,9,9] than for [10,-100,-100].

[0]:
```python
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
```

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

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



---

2 IMPORTANT

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

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

[0]:
```
import os

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

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

[0]:

two_layer_net

April 23, 2020

[0]:
```
from google.colab import drive

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

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

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

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

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

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-22 16:20:24--  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  49.2MB/s    in 3.6s

2020-04-22 16:20:27 (45.0 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

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

1 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.

[0]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

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

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

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

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to rep-

resent 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.

[0]:
```
# 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.

[0]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()
```

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

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

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

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

3 Forward pass: compute loss

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

[0]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

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

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

4 Backward pass

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

[0]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

```
W2 max relative error: 3.440708e-09
W1 max relative error: 3.561318e-09
b2 max relative error: 4.447625e-11
b1 max relative error: 2.738420e-09
```

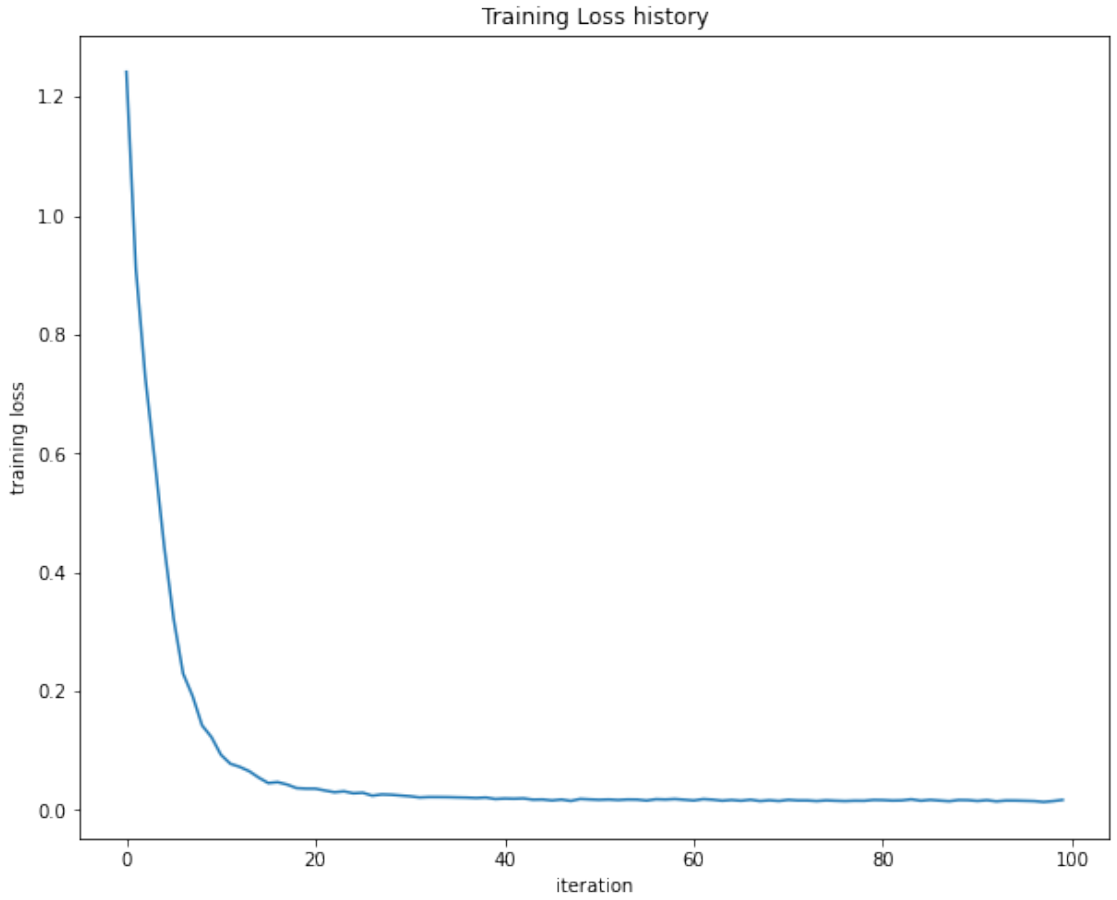
5 Train the network

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

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

[0]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.017149607938732093
```



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.

[0]:
```
from cs231n.data_utils import load_CIFAR10

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

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

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

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

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

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

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

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


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

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

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

7 Train a network

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

[0]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

```
iteration 0 / 1000: loss 2.302970
iteration 100 / 1000: loss 2.302474
iteration 200 / 1000: loss 2.297076
iteration 300 / 1000: loss 2.257328
iteration 400 / 1000: loss 2.230484
iteration 500 / 1000: loss 2.150620
iteration 600 / 1000: loss 2.080736
iteration 700 / 1000: loss 2.054914
iteration 800 / 1000: loss 1.979290
iteration 900 / 1000: loss 2.039101
Validation accuracy:  0.287
```

8 Debug the training

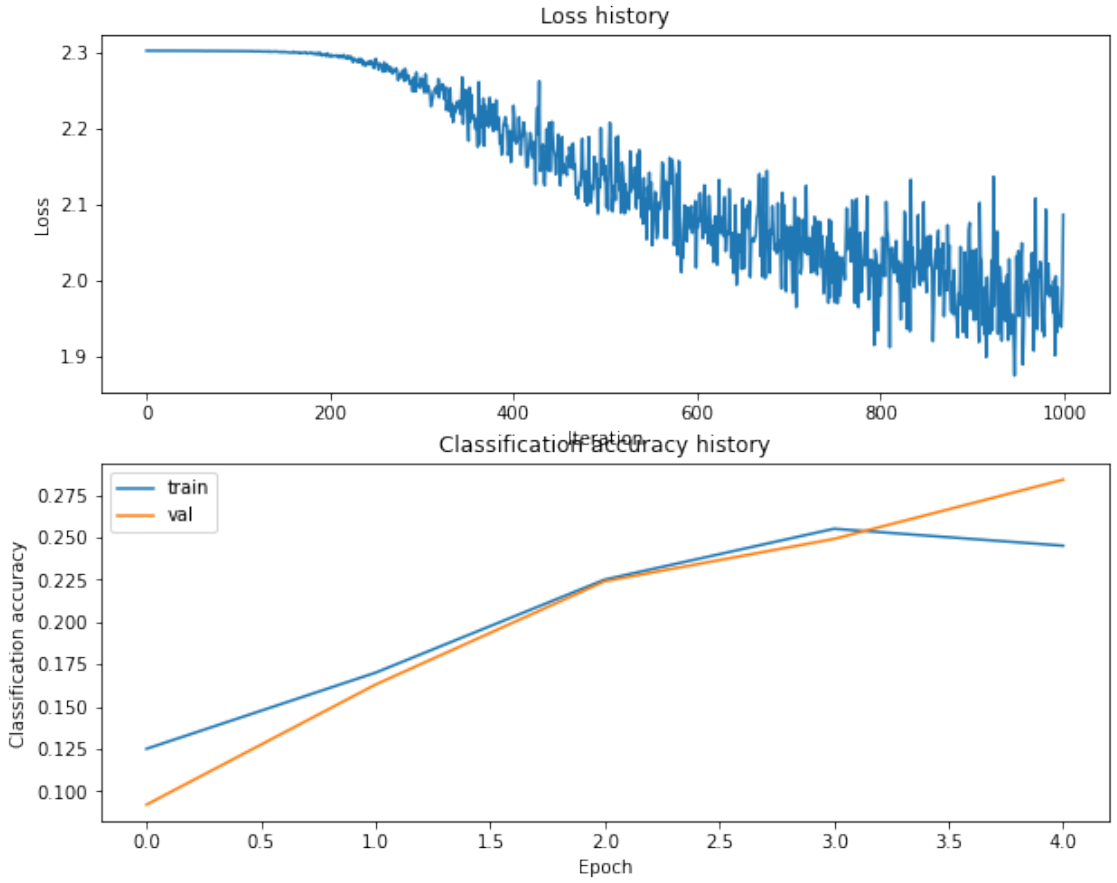
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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

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

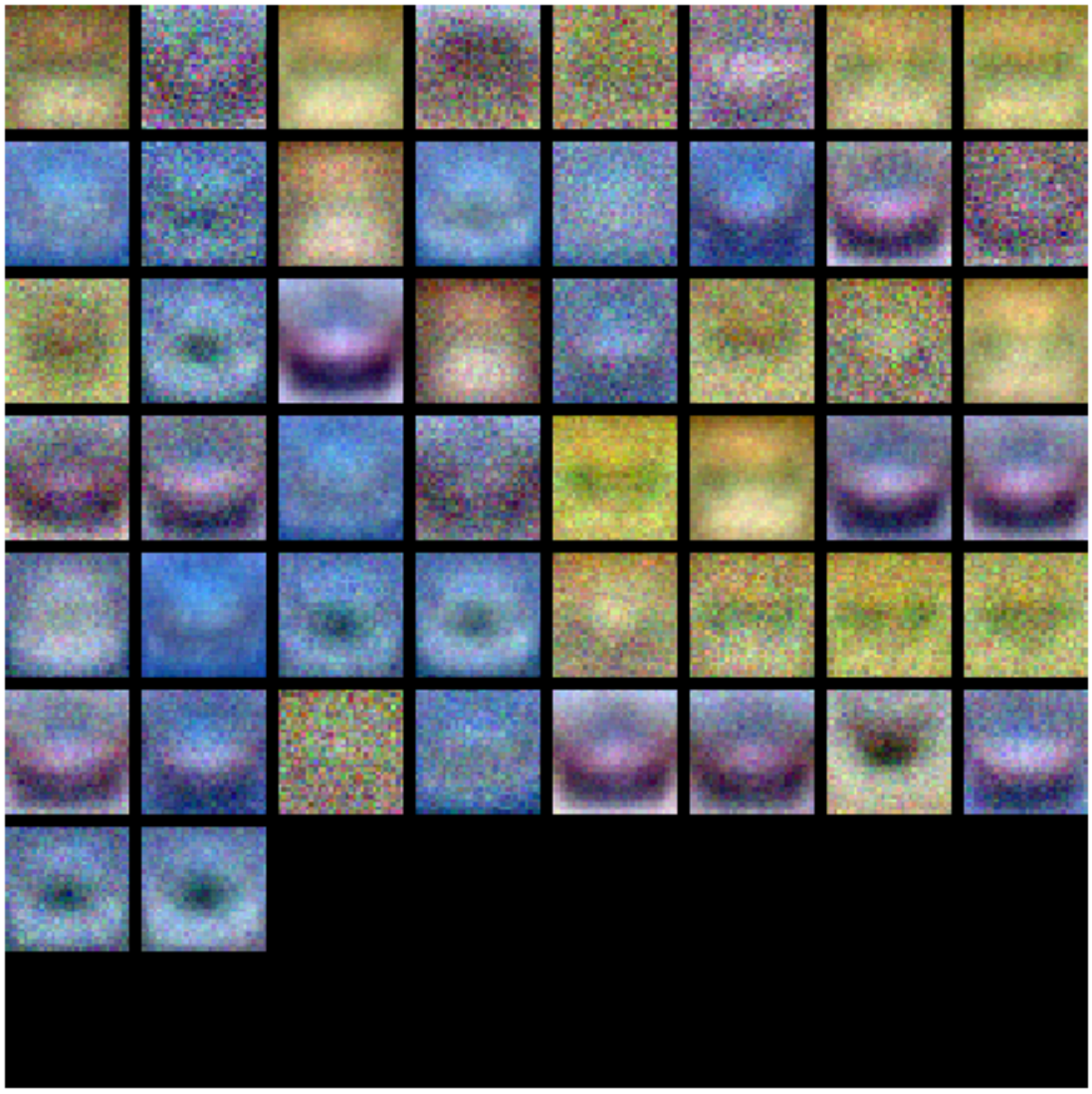
[0]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
```

```
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()
```



[0]:
```
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 : I set up different learning_rates, regularization_strengths, num_iters and hidden_sizes. Then iterated through all the combinations and trained the model. I have done a little bit tweak for learning_rate and regularization_strength. For other hyperparameters, such as num_iters and hidden_sizes, I just increased the iteration numbers and hidden layer size. After tunning hyperparameter, the model can reach around 55% accuracy on validation set.

[0]:
```
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)*****
```

```
results = {}
best_val = -1
input_size = 32 * 32 * 3
num_classes = 10
batch_size=200
learning_rate_decay=0.95

learning_rates = [1e-3,5e-3]
regularization_strengths = [5e-2,5e-4,5e-6]
num_iters_ = [3000,4000,5000]
hidden_sizes = [50,100,150,200,250]
#Params:  0.001 0.0005 5000 250
#Validation accuracy:  0.555

for learning_rate in learning_rates:
  for reg in regularization_strengths:
    for num_iters in num_iters_:
      for hidden_size in hidden_sizes:
        net = TwoLayerNet(input_size, hidden_size, num_classes)
        stats = net.train(X_train, y_train, X_val, y_val,␣
↪learning_rate,learning_rate_decay,reg,num_iters,batch_size,verbose=False)
        train_acc = np.max(stats['train_acc_history'])
        val_acc = np.max(stats['val_acc_history'])
        print('Params: ',learning_rate,reg,num_iters,hidden_size)
        print('Validation accuracy: ', val_acc)

        if best_val < val_acc:
          best_val = val_acc
          best_net = net
        results[(learning_rate, reg)] = (train_acc,val_acc)

best_val_acc = (best_net.predict(X_val) == y_val).mean()
print('Best Validation accuracy: ', best_val_acc)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
Params:  0.001 0.05 3000 50
Validation accuracy:  0.509
Params:  0.001 0.05 3000 100
Validation accuracy:  0.531
Params:  0.001 0.05 3000 150
Validation accuracy:  0.54
Params:  0.001 0.05 3000 200
Validation accuracy:  0.546
Params:  0.001 0.05 3000 250
Validation accuracy:  0.533
Params:  0.001 0.05 4000 50
Validation accuracy:  0.512
```

```
Params:  0.001 0.05 4000 100
Validation accuracy:  0.533
Params:  0.001 0.05 4000 150
Validation accuracy:  0.547
Params:  0.001 0.05 4000 200
Validation accuracy:  0.539
Params:  0.001 0.05 4000 250
Validation accuracy:  0.555
Params:  0.001 0.05 5000 50
Validation accuracy:  0.531
Params:  0.001 0.05 5000 100
Validation accuracy:  0.541
Params:  0.001 0.05 5000 150
Validation accuracy:  0.539
Params:  0.001 0.05 5000 200
Validation accuracy:  0.543
Params:  0.001 0.05 5000 250
Validation accuracy:  0.549
Params:  0.001 0.0005 3000 50
Validation accuracy:  0.515
Params:  0.001 0.0005 3000 100
Validation accuracy:  0.54
Params:  0.001 0.0005 3000 150
Validation accuracy:  0.533
Params:  0.001 0.0005 3000 200
Validation accuracy:  0.533
Params:  0.001 0.0005 3000 250
Validation accuracy:  0.546
Params:  0.001 0.0005 4000 50
Validation accuracy:  0.518
Params:  0.001 0.0005 4000 100
Validation accuracy:  0.547
Params:  0.001 0.0005 4000 150
Validation accuracy:  0.537
Params:  0.001 0.0005 4000 200
Validation accuracy:  0.546
Params:  0.001 0.0005 4000 250
Validation accuracy:  0.539
Params:  0.001 0.0005 5000 50
Validation accuracy:  0.528
Params:  0.001 0.0005 5000 100
Validation accuracy:  0.543
Params:  0.001 0.0005 5000 150
Validation accuracy:  0.548
Params:  0.001 0.0005 5000 200
Validation accuracy:  0.55
Params:  0.001 0.0005 5000 250
Validation accuracy:  0.555
```

```
Params:  0.001 5e-06 3000 50
Validation accuracy:  0.516
Params:  0.001 5e-06 3000 100
Validation accuracy:  0.529
Params:  0.001 5e-06 3000 150
Validation accuracy:  0.52
Params:  0.001 5e-06 3000 200
Validation accuracy:  0.561
Params:  0.001 5e-06 3000 250
Validation accuracy:  0.531
Params:  0.001 5e-06 4000 50
Validation accuracy:  0.517
Params:  0.001 5e-06 4000 100
Validation accuracy:  0.538
Params:  0.001 5e-06 4000 150
Validation accuracy:  0.539
Params:  0.001 5e-06 4000 200
Validation accuracy:  0.549
Params:  0.001 5e-06 4000 250
Validation accuracy:  0.543
Params:  0.001 5e-06 5000 50
Validation accuracy:  0.513
Params:  0.001 5e-06 5000 100
Validation accuracy:  0.529
Params:  0.001 5e-06 5000 150
Validation accuracy:  0.54
Params:  0.001 5e-06 5000 200
Validation accuracy:  0.554
Params:  0.001 5e-06 5000 250
Validation accuracy:  0.543

/content/cs231n/classifiers/neural_net.py:107: RuntimeWarning: divide by zero
encountered in log
  loss = np.sum(-np.log(p[range(N),y]))
/content/cs231n/classifiers/neural_net.py:103: RuntimeWarning: overflow
encountered in subtract
  scores -= np.max(scores,axis=1,keepdims=True)
/content/cs231n/classifiers/neural_net.py:103: RuntimeWarning: invalid value
encountered in subtract
  scores -= np.max(scores,axis=1,keepdims=True)
/content/cs231n/classifiers/neural_net.py:127: RuntimeWarning: invalid value
encountered in greater
  dh1 = (h1>0) * dz #(N,H)*(N,H) = (N,H)

Params:  0.005 0.05 3000 50
Validation accuracy:  0.202
Params:  0.005 0.05 3000 100
Validation accuracy:  0.186
```

```
Params:  0.005 0.05 3000 150
Validation accuracy:  0.184
Params:  0.005 0.05 3000 200
Validation accuracy:  0.177
Params:  0.005 0.05 3000 250
Validation accuracy:  0.197
Params:  0.005 0.05 4000 50
Validation accuracy:  0.196
Params:  0.005 0.05 4000 100
Validation accuracy:  0.16
Params:  0.005 0.05 4000 150
Validation accuracy:  0.162
Params:  0.005 0.05 4000 200
Validation accuracy:  0.173
Params:  0.005 0.05 4000 250
Validation accuracy:  0.145
Params:  0.005 0.05 5000 50
Validation accuracy:  0.134
Params:  0.005 0.05 5000 100
Validation accuracy:  0.156
Params:  0.005 0.05 5000 150
Validation accuracy:  0.149
Params:  0.005 0.05 5000 200
Validation accuracy:  0.193
Params:  0.005 0.05 5000 250
Validation accuracy:  0.165
Params:  0.005 0.0005 3000 50
Validation accuracy:  0.16
Params:  0.005 0.0005 3000 100
Validation accuracy:  0.158
Params:  0.005 0.0005 3000 150
Validation accuracy:  0.216
Params:  0.005 0.0005 3000 200
Validation accuracy:  0.179
Params:  0.005 0.0005 3000 250
Validation accuracy:  0.164
Params:  0.005 0.0005 4000 50
Validation accuracy:  0.156
Params:  0.005 0.0005 4000 100
Validation accuracy:  0.157
Params:  0.005 0.0005 4000 150
Validation accuracy:  0.179
Params:  0.005 0.0005 4000 200
Validation accuracy:  0.18
Params:  0.005 0.0005 4000 250
Validation accuracy:  0.176
Params:  0.005 0.0005 5000 50
Validation accuracy:  0.21
```

```
Params:  0.005 0.0005 5000 100
Validation accuracy:  0.164
Params:  0.005 0.0005 5000 150
Validation accuracy:  0.166
Params:  0.005 0.0005 5000 200
Validation accuracy:  0.213
Params:  0.005 0.0005 5000 250
Validation accuracy:  0.223
Params:  0.005 5e-06 3000 50
Validation accuracy:  0.171
Params:  0.005 5e-06 3000 100
Validation accuracy:  0.148
Params:  0.005 5e-06 3000 150
Validation accuracy:  0.167
Params:  0.005 5e-06 3000 200
Validation accuracy:  0.171
Params:  0.005 5e-06 3000 250
Validation accuracy:  0.19
Params:  0.005 5e-06 4000 50
Validation accuracy:  0.142
Params:  0.005 5e-06 4000 100
Validation accuracy:  0.191
Params:  0.005 5e-06 4000 150
Validation accuracy:  0.178
Params:  0.005 5e-06 4000 200
Validation accuracy:  0.225
Params:  0.005 5e-06 4000 250
Validation accuracy:  0.187
Params:  0.005 5e-06 5000 50
Validation accuracy:  0.185
Params:  0.005 5e-06 5000 100
Validation accuracy:  0.153
Params:  0.005 5e-06 5000 150
Validation accuracy:  0.197
Params:  0.005 5e-06 5000 200
Validation accuracy:  0.174
Params:  0.005 5e-06 5000 250
Validation accuracy:  0.191
Best Validation accuracy:  0.55
```

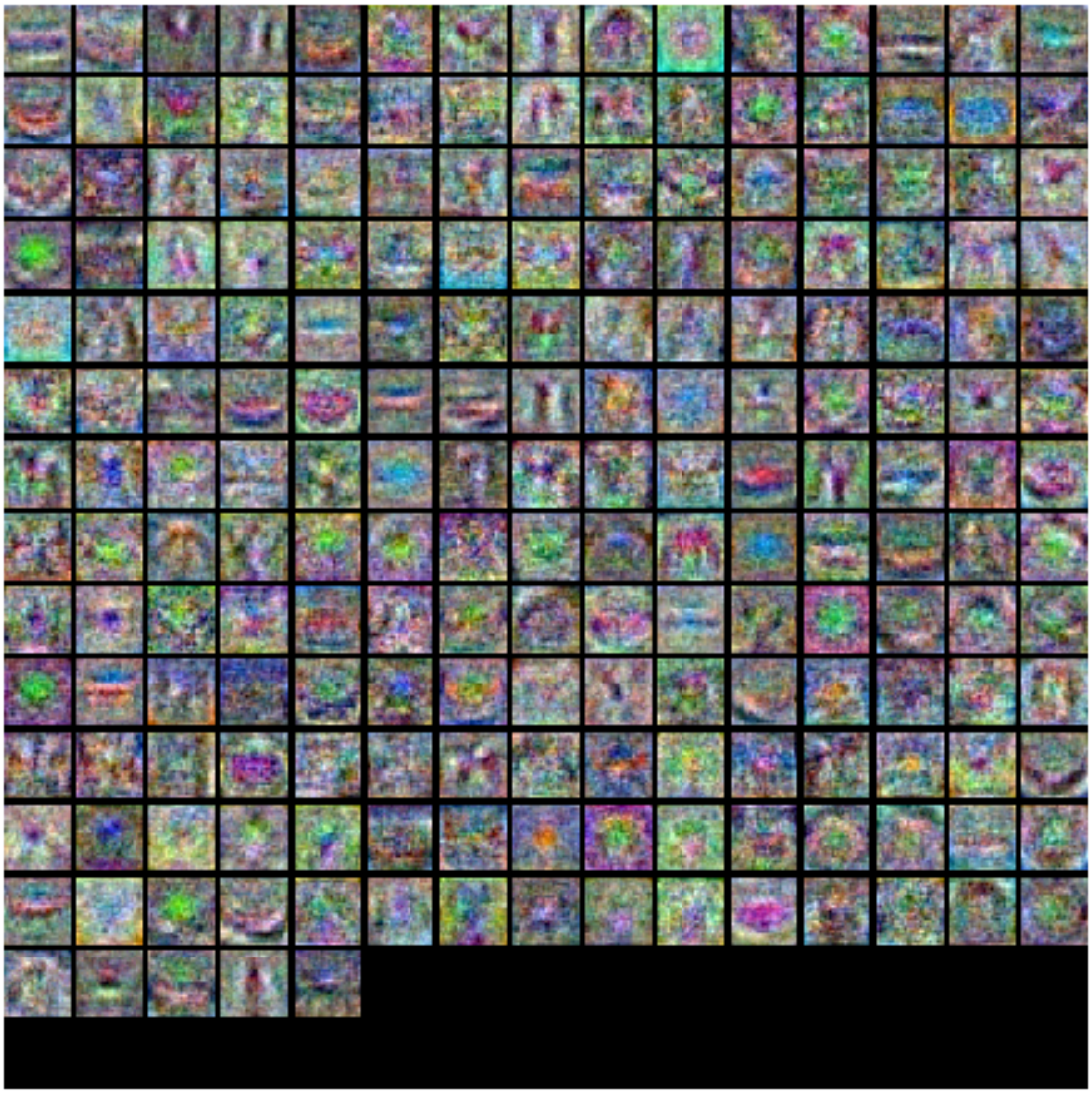
[0]:
```
print(results)
```

```
{(0.001, 0.05): (0.8, 0.549), (0.001, 0.0005): (0.805, 0.555), (0.001, 5e-06):
(0.82, 0.543), (0.005, 0.05): (0.265, 0.165), (0.005, 0.0005): (0.395, 0.223),
(0.005, 5e-06): (0.325, 0.191)}
```

[0]:
```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.55
```

[0]:
```
# 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%.

[0]:
```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.517
```

Inline Question

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

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

YourAnswer : 1 & 3

YourExplanation : \ When testing accuracy is much lower than the training accuracy shows that the model might overfit the data, so called high variance. Training on more data and increasing the regularization strength should be helpful.

---

11 IMPORTANT

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

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

[0]:
```
import os

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

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

[0]:
```
```

features

April 23, 2020

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

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

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

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

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

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

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-22 21:51:43--  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  15.9MB/s    in 11s

2020-04-22 21:51:55 (14.3 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
```

1 Image features exercise

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

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

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

[0]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


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

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

1.1 Load data

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

[0]:
```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

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

1.2 Extract Features

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

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

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

matrix where each column is the concatenation of all feature vectors for a single image.

[4]:
```
from cs231n.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

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

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

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

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

```
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

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

[7]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-2, 1e-9, 1e-8, 1e-7]
regularization_strengths = [0.1, 5e4, 5e5, 5e6]
#lr 1.000000e-02 reg 1.000000e-01 train accuracy: 0.504449 val accuracy: 0.
 ↪482000
results = {}
best_val = -1
best_svm = None
```

```
################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for learning_rate in learning_rates:
  for regularization_strength in regularization_strengths:
    svm = LinearSVM()
    loss_hist = svm.train(X_train_feats, y_train, learning_rate,
 ↪regularization_strength, num_iters=4000, verbose=False)

    y_train_pred = svm.predict(X_train_feats)
    train_acc = np.mean(y_train_pred==y_train)
    y_val_pred = svm.predict(X_val_feats)
    val_acc = np.mean(y_val_pred==y_val)
    if best_val < val_acc:
      best_val = val_acc
      best_svm = svm
    results[(learning_rate, regularization_strength)] = (train_acc,val_acc)

# *****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)
```

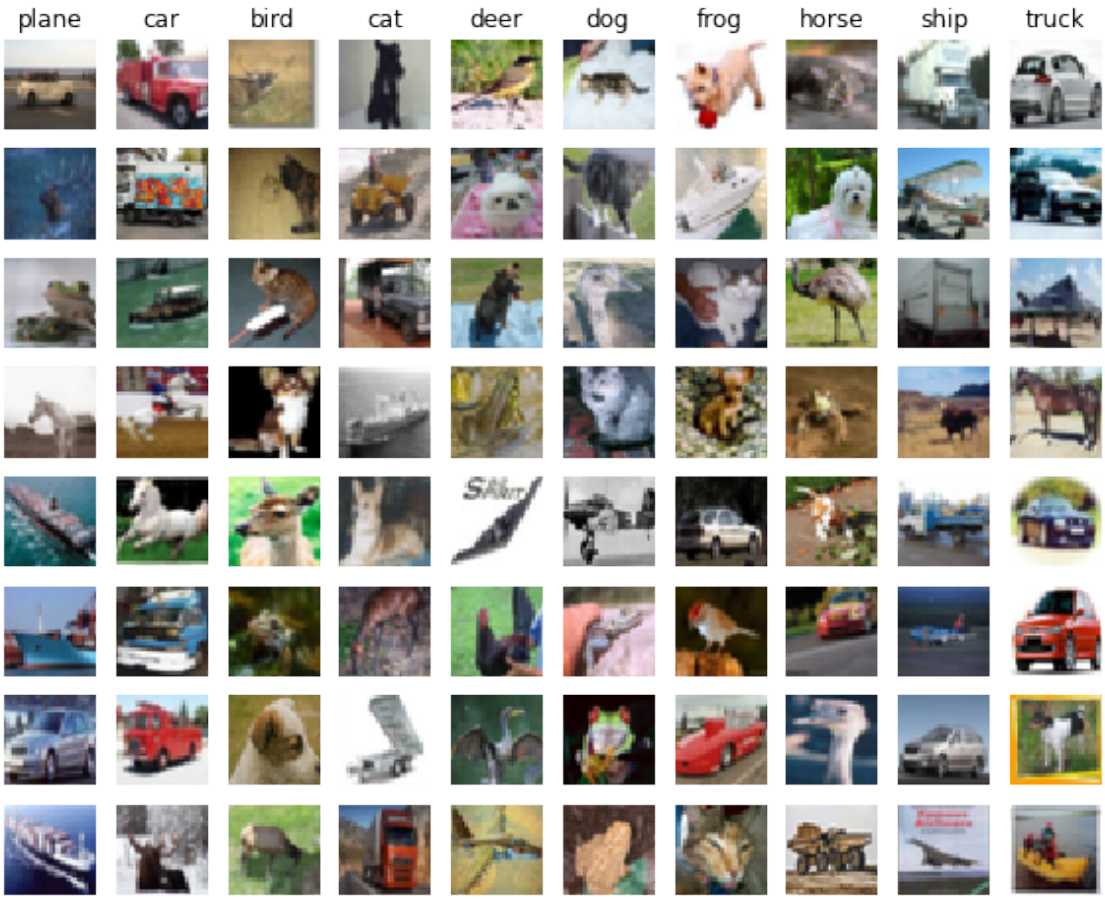
```
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/cs231n/classifiers/linear_svm.py:90: RuntimeWarning: overflow
encountered in multiply
```

```
  loss += reg * np.sum(W * W)
/content/cs231n/classifiers/linear_svm.py:108: RuntimeWarning: overflow
encountered in multiply
  dW += 2*reg * W
/content/cs231n/classifiers/linear_svm.py:85: RuntimeWarning: invalid value
encountered in subtract
  margin = scores - correct_class_score.reshape(num_train,1) + 1 #(N,C)
/content/cs231n/classifiers/linear_svm.py:87: RuntimeWarning: invalid value
encountered in greater
  margin = (margin>0) * margin
/content/cs231n/classifiers/linear_svm.py:87: RuntimeWarning: invalid value
encountered in multiply
  margin = (margin>0) * margin
/content/cs231n/classifiers/linear_svm.py:103: RuntimeWarning: invalid value
encountered in greater
  margin[margin>0] = 1 #(N,C)
/content/cs231n/classifiers/linear_svm.py:90: RuntimeWarning: overflow
encountered in double_scalars
  loss += reg * np.sum(W * W)

lr 1.000000e-09 reg 1.000000e-01 train accuracy: 0.097490 val accuracy: 0.090000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.105531 val accuracy: 0.101000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.117061 val accuracy: 0.145000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414429 val accuracy: 0.420000
lr 1.000000e-08 reg 1.000000e-01 train accuracy: 0.108939 val accuracy: 0.132000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.133347 val accuracy: 0.146000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.413980 val accuracy: 0.412000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.403490 val accuracy: 0.398000
lr 1.000000e-07 reg 1.000000e-01 train accuracy: 0.155367 val accuracy: 0.170000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.414857 val accuracy: 0.415000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.404388 val accuracy: 0.396000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.340408 val accuracy: 0.349000
lr 1.000000e-02 reg 1.000000e-01 train accuracy: 0.504449 val accuracy: 0.482000
lr 1.000000e-02 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-02 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-02 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.482000
```

[8]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
 ↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.484
```

[9]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

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



1.3.1 Inline question 1:

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

Your Answer : \ Yes. Some of the misclassification results make sense. Since HOG captures the texture of image while ignoring color information, and and the color histogram represents the color of the input image while ignoring texture. For example, the model misclassified ships to plane, because of the blue color of sea is same as sky. The model mixed up different kinds of animal, such as dog, cat, deer, and horse, etc. since they are all have 4 legs, fur and tail. For truck class, there are many cars have been misclassified. For ship class, silver cars and grey planes have been misclassified to it, those colors are similar. However, I think HOG and color histogram is not good enough to help make a high accuracy classifier.

1.4 Neural Network on image features

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

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

[10]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

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

[17]:
```
from cs231n.classifiers.neural_net import TwoLayerNet

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

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

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    ␣
 ↪#
```

```
# cross-validate various parameters as in previous sections. Store your best
 ↪#
# model in the best_net variable.
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

batch_size=200
learning_rate_decay=0.95

learning_rates = [0.1,0.2,1e-2,1e-3]
regularization_strengths = [1e-2,1e-4,1e-6,1e-7]
num_iters = 5000


for learning_rate in learning_rates:
  for reg in regularization_strengths:
    stats = net.
 ↪train(X_train_feats,y_train,X_val_feats,y_val,learning_rate,learning_rate_decay,reg,num_ite
    train_acc = np.max(stats['train_acc_history'])
    val_acc = np.max(stats['val_acc_history'])
    print('Params: ',learning_rate,reg,num_iters)
    print('Validation accuracy: ', val_acc)

    if best_val < val_acc:
      best_val = val_acc
      best_net = net
    results[(learning_rate, reg)] = (train_acc,val_acc)

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

```
Params:  0.1 0.01 5000
Validation accuracy:  0.521
Params:  0.1 0.0001 5000
Validation accuracy:  0.613
Params:  0.1 1e-06 5000
Validation accuracy:  0.621
Params:  0.1 1e-07 5000
Validation accuracy:  0.594
Params:  0.2 0.01 5000
Validation accuracy:  0.573
Params:  0.2 0.0001 5000
Validation accuracy:  0.608
Params:  0.2 1e-06 5000
Validation accuracy:  0.595
Params:  0.2 1e-07 5000
Validation accuracy:  0.591
```

```
Params:  0.01 0.01 5000
Validation accuracy:  0.6
Params:  0.01 0.0001 5000
Validation accuracy:  0.609
Params:  0.01 1e-06 5000
Validation accuracy:  0.605
Params:  0.01 1e-07 5000
Validation accuracy:  0.604
Params:  0.001 0.01 5000
Validation accuracy:  0.603
Params:  0.001 0.0001 5000
Validation accuracy:  0.602
Params:  0.001 1e-06 5000
Validation accuracy:  0.602
Params:  0.001 1e-07 5000
Validation accuracy:  0.602
```

[18]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

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

```
0.598
```

---

2 IMPORTANT

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

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

[0]:
```
import os

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

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

[0]:
```
```