knn

April 23, 2020

[20]:
```
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 ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-23 04:41:28--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  58.2MB/s    in 2.8s

2020-04-23 04:41:31 (58.2 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

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

1 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.

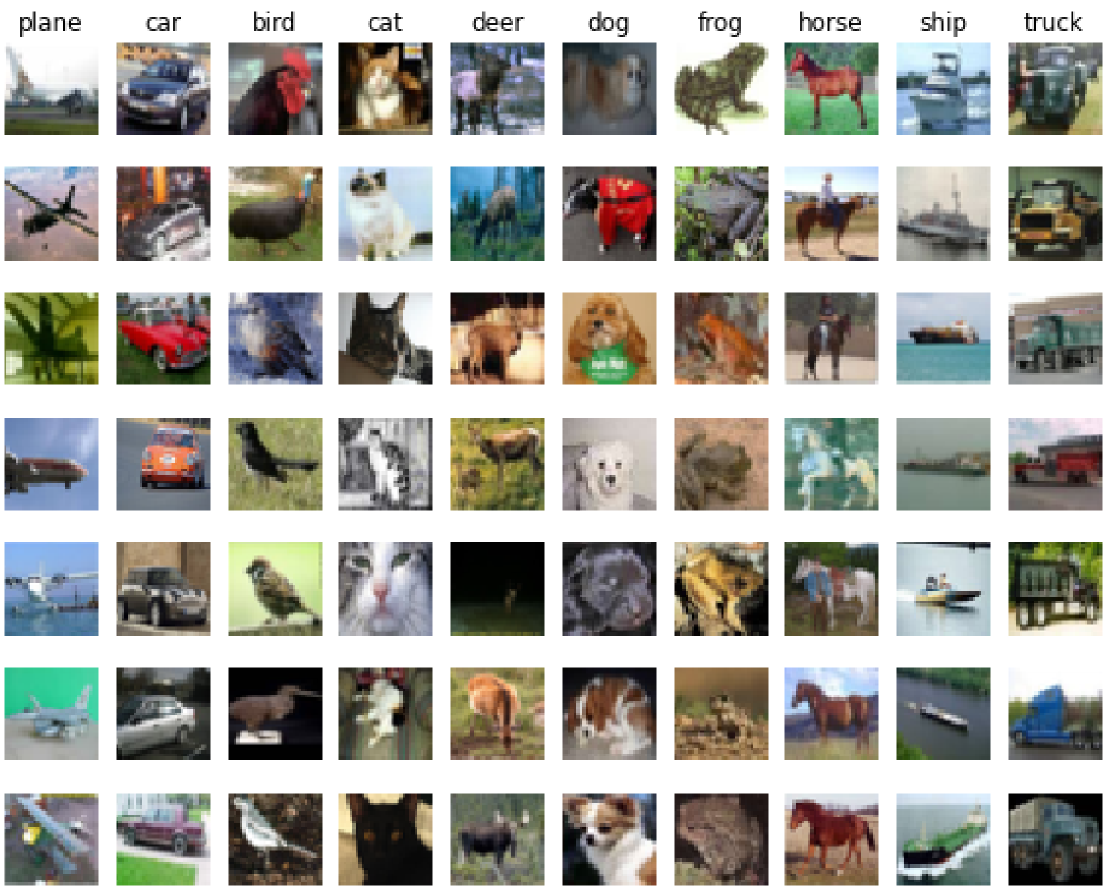
[21]:
```
# Run some setup code for this notebook.

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
```

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

[22]:
```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[23]:
```python
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[24]:
```python
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

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

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

[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)
```

[26]:
```
from google.colab import drive
drive.mount('/content/drive')
```

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

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.

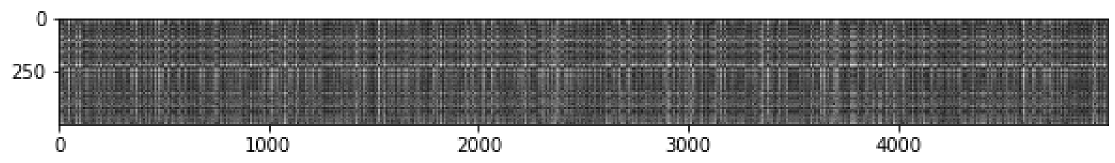
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[27]:
```
# 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)
```

[28]:
```
# 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 : Since we are working with pixel intensity values, distinctly bright rows indicate a significant delta in pixel intensities between the corresponding test and training images (say a black background being "compared" to a white background and vice-versa). While foreground differences can obviously contribute to pixel deltas, in some cases the background of an image can be a relatively huge proportion of the image which can heavily contribute to a significant pixel-level delta. For the case of distinctly bright rows, we can infer that for the particular test image corresponding to that row, the test images' contents have a distinctly different foreground/background that leads to a significantly large pixel delta. Similarly, the distinctly bright columns can be due to a particular train image having content that does not match the test images.

For e.g., consider an image which has a cat in the center of the image on a white background. If the other images that this image is being compared against have black backgrounds, this will cause a large pixel-wise distance between the two images.

[29]:
```
# 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`:

[30]:
```
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, 4

YourExplanation :

1. Subtracting the mean μ from the data does have the conditioning effect of centering the data around the origin. However, in our case since we are dealing with images, the feature ranges are aleady localized, i.e., all pixels lie within [0-255]. Subtracting the mean will only offset the pixel values and will likely not have a drastic effect on the performance of the Nearest Neighbour classifier. To demonstrate this mathematically, the distance between a given test sample over each sample in the training set is given by,

$$L_1 = \frac{1}{n} \sum_{k=1}^{n} || \, (x_{test} - \mu) - (x_{train}^{(k)} - \mu) \, ||_1 = || \, x_{test} - x_{train}^{(k)} \, ||_1$$

Here, x_{test} and x_{train}^{k} are vectors that hold a test and the k^{th} training image.

2. Subtracting the per-pixel mean μ_{ij} from the data also has the conditioning effect of centering the data around the origin. However, as explained above, since all pixel values are already localized to [0-255], it wouldn't impact the performance of the classifier. Similar to (1) above, to demonstrate this mathematically, the distance between a test and k^{th} training sample is given by,

$$L_1 = \frac{1}{nhw} \sum_{k=1}^{n} \sum_{i=1}^{h} \sum_{j=1}^{w} || \, (p_{ij} - \mu_{ij}) - (q_{ij}^{(k)} - \mu_{ij}) \, ||_1 = || \, p_{ij} - q_{ij}^{(k)} \, ||_1$$

Here, p_{ij} and $q_{ij}^{(k)}$ are the pixels at (i, j) within the test image and the k^{th} training image respectively.

3. Subtracting the mean and dividing by the standard deviation would yield zero-centered data (μ = 0) with unit variance (σ = 1), i.e., properties of a standard normal distribution. This pre-processing step would normally be very useful if different features/dimensions of the data had different ranges, e.g., if feature #1 has a range [-1000-1000] and feature #2 has a

range of [0-1], feature #1 will have more impact in the calculation of distances because of the order of magnitude difference between the two features. In such cases where there is a stark delta in the features ranges, normalization is key. However, in our case since we are dealing with images, the feature ranges are mostly similar, i.e., all pixels lie within [0-255] and there is no feature scale mismatch. The benefits of normalization are limited in our specific case due to the fact that we have an image dataset. Performing feature scaling does help facilitate learning by bounding the gradients and thus making gradient descent converge faster and speed up training, but this does not affect classifier accuracy. Applying normalization thus does not affect the performance of the classifier.

4. Similar to above, since we are dealing with images, the feature ranges are mostly similar, i.e., all pixels lie within [0-255]. Performing feature scaling does help facilitate learning by bounding the gradients and thus making gradient descent converge faster but this does not improve classifier accuracy. Thus, pixel-wise normalization would not improve performance during training.

5. L1 distance is not invariant to the rotation of the coordinate axes, unlike L2. The distance between points changes with the rotation of axes and thus, the performance of the classifier can potentially be affected. To demonstrate this mathematically, consider that the coordinate axes of the data (in red) are rotated 45 degrees (or by pi/4 radians) in the counter-clockwise direction (in blue), illustrated in the diagram below.

Lets consider three points x = (0, 1), y = (1, 0) and z = (1, -2). The L1-distance between the three points with y as pivot would be,

$$\|x - y\|_1 = \|y - z\|_1 = 2$$

This implies that both x and z are at the same distance from y. Now consider the 45 degrees rotation matrix:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

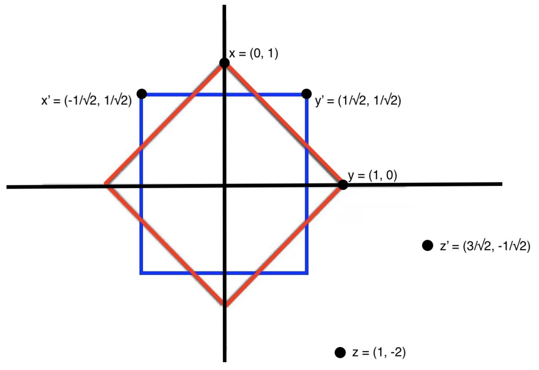
Then, the new co-ordinates after the 45 degrees rotation would be given by,

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

Thus, x, y and z get transformed to x′, y′ and z′ as follows,



$$\begin{aligned}
x' &= Ax \\
&= \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \\
y' &= Ay \\
&= \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \\
z' &= Az \\
&= \begin{bmatrix} \frac{3}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}
\end{aligned}$$

The new L1-distance between the three points (again, with y as pivot) is,

$$\begin{aligned}
\|x' - y'\|_1 &= \sqrt{2} \\
\|y' - z'\|_1 &= \frac{3}{\sqrt{2}}
\end{aligned}$$

and thus $\|x' - y'\|_1 < \|y' - z'\|_1$. Hence, ordering is not preserved with L1 distance post rotation.

[31]:
```
# 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
```

[32]:
```
# 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
```

[33]:
```
# 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 35.387777 seconds
One loop version took 30.430853 seconds
No loop version took 0.516027 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.

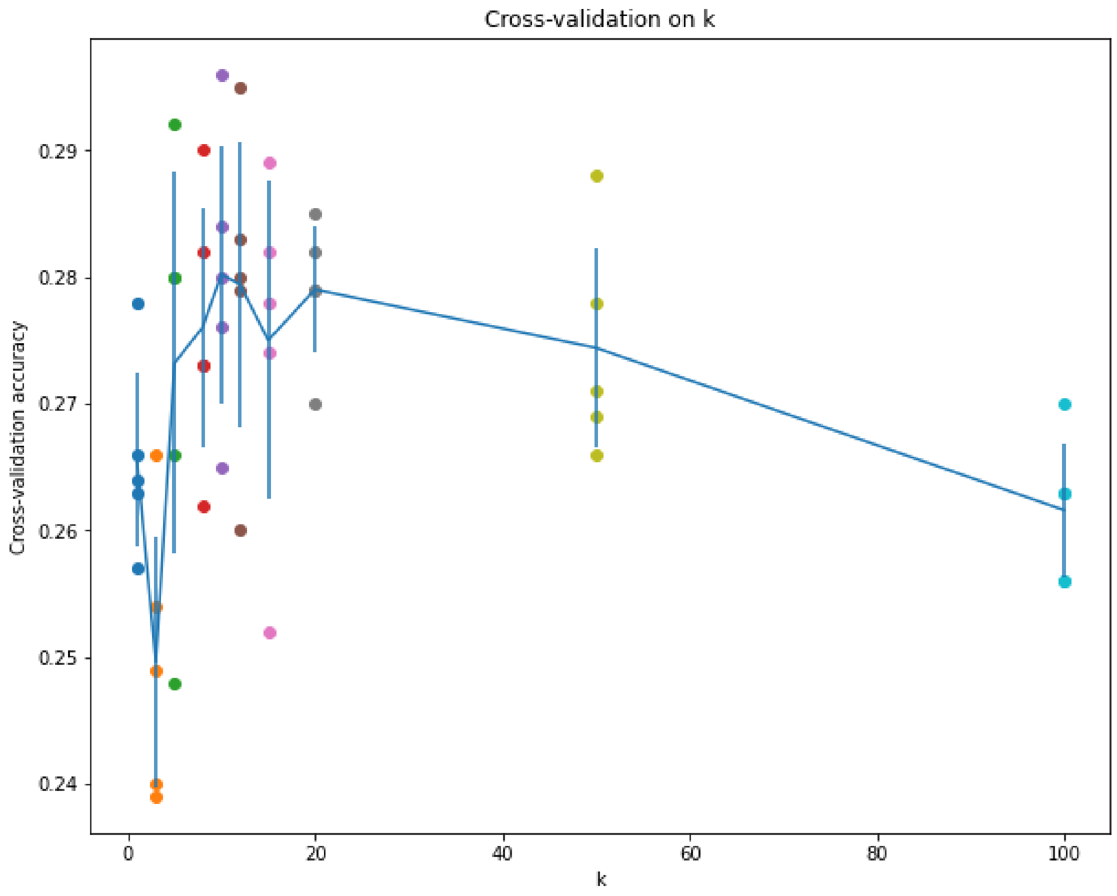
[34]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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

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

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

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


################################################################################
# TODO:                                                                        ␣
↪#
# Perform k-fold cross validation to find the best value of k. For each       ␣
↪#
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,  ␣
↪#
# where in each case you use all but one of the folds as training data and the␣
↪#
# last fold as a validation set. Store the accuracies for all fold and all    ␣
↪#
```

```
# values of k in the k_to_accuracies dictionary.                               ␣
↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for current_k in k_choices:
    k_to_accuracies[current_k] = []

    for validation_fold in range(num_folds):
        classifier = KNearestNeighbor() # create a new classifier object

        # concatenate together all other folds in X_train_folds apart from the␣
↪validation fold
        # and train the classifier
        classifier.train(np.concatenate(X_train_folds[:validation_fold] +␣
↪X_train_folds[validation_fold + 1:]),
                         np.concatenate(y_train_folds[:validation_fold] +␣
↪y_train_folds[validation_fold + 1:]))

        y_pred_fold = classifier.predict(X_train_folds[validation_fold], k =␣
↪current_k)

        k_to_accuracies[current_k].append(np.mean(y_pred_fold ==␣
↪y_train_folds[validation_fold]))

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

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

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

```
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

[35]:
```
# 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()
```



[36]:
```
# 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 = k_choices[accuracies_mean.argmax()]

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

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

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

Inline Question 3

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

$Your Answer$: 2 and 4.

$Your Explanation$:

1. False. The decision boundary of the k-NN classifier is not linear. If you consider a dataset where the classes belong to concentric circles, the decision boundaries in this case will follow the curvature of the concentric circles.
2. True. The training error of a 1-NN will always be lower than that of 5-NN because for each training example, its nearest neighbor is always going to be itself, i.e., error of 1-NN will be zero.
3. False. The test error of a 1-NN will not always be lower than 5-NN. Lets consider an example. Suppose $x_{train} = (1, 2, 3, 4, 5)$ and $y_{train} = (1, 0, 0, 0, 0)$. For a test sample of $x = 0$ with $y = 0$, y_{pred} would be 1 for 1-NN (thus, error = 100%) while y_{pred} would be 0 for 5-NN (thus, error = 0%). The value of k is thus data-dependent, which is why we need to perform cross validation to determine the best k for your intended application and dataset.
4. True. The testing phase of k-NN is essentially performing comparisons of each test sample with the entire training set, which needs one full pass through the training set. Infact, the training phase of k-NN, which consists of remembering the traning set would also grow with the size of the training set. However, in order to decrease the number of comparisons and thus improve time complexity, we can use Approximate Nearest Neighbor techniques (such as k-d trees, ball trees etc.).

---

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()))
```

svm

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 ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-19 08:28:04--  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  46.6MB/s    in 3.8s

2020-04-19 08:28:08 (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
```

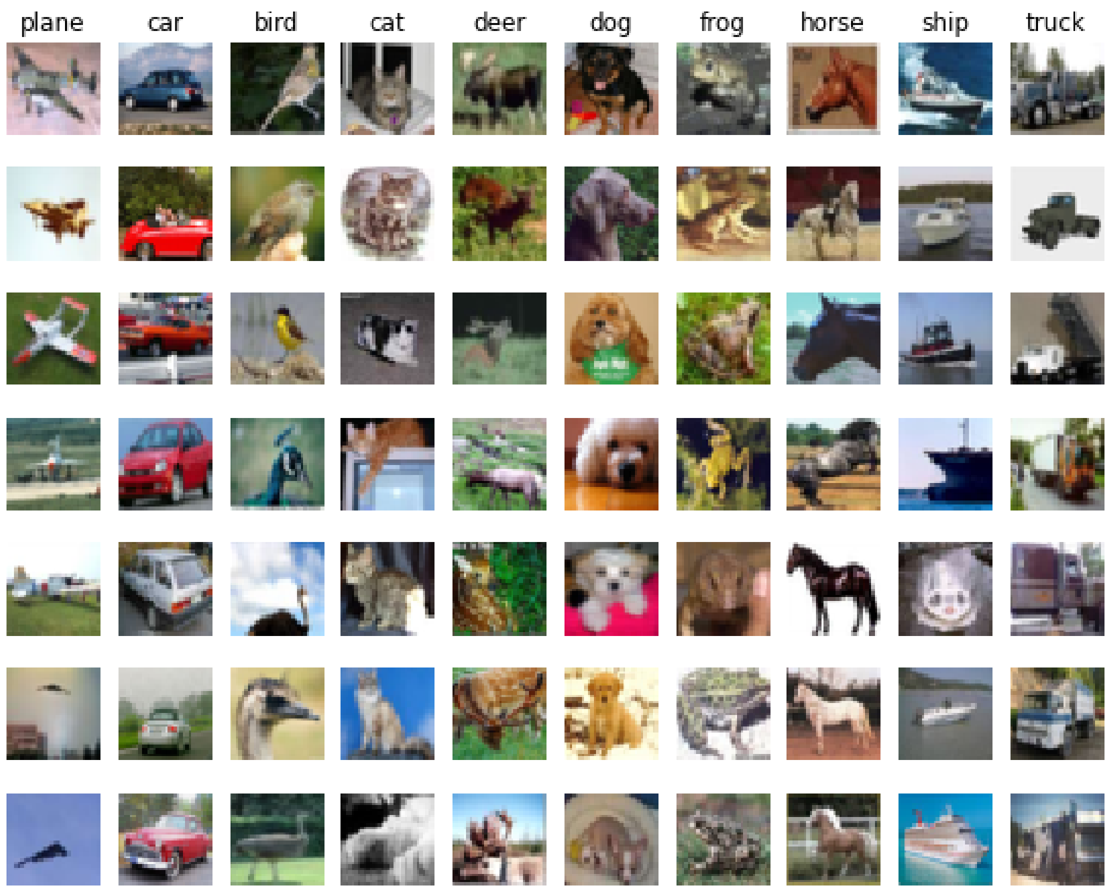
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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)
```

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[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]
```

```python
# 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,)
```

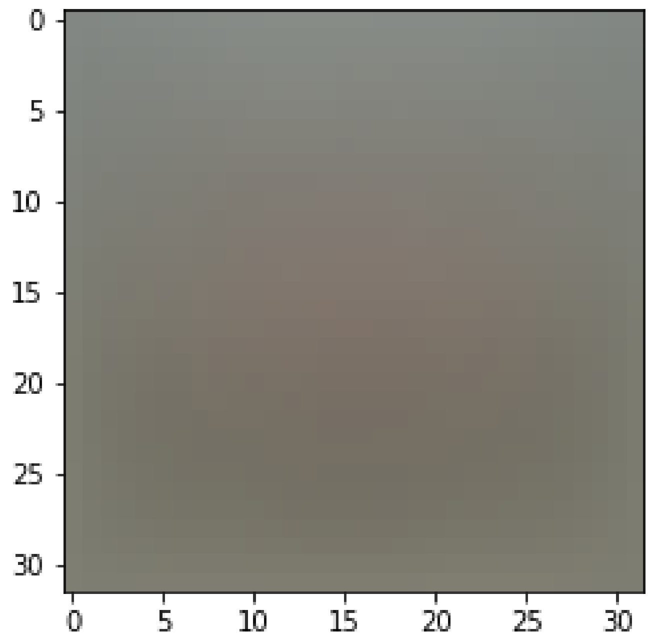
[0]:
```python
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

[0]:
```python
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
```

```python
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

The loss function for a SVM loss is given by:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + 1)$$

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: 8.974389
```

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: -9.571098 analytic: -9.571098, relative error: 1.515400e-11
numerical: 25.058381 analytic: 25.058381, relative error: 8.780704e-12
numerical: 26.673645 analytic: 26.673645, relative error: 1.396251e-11
numerical: 24.445590 analytic: 24.445590, relative error: 2.285221e-11
numerical: 24.340310 analytic: 24.298053, relative error: 8.688018e-04
numerical: 24.188987 analytic: 24.188987, relative error: 3.314928e-12
numerical: -15.150728 analytic: -15.124028, relative error: 8.819086e-04
numerical: 13.356248 analytic: 13.356248, relative error: 7.883734e-12
numerical: -10.771455 analytic: -10.771455, relative error: 1.824420e-11
numerical: 3.441360 analytic: 3.441360, relative error: 1.092105e-12
numerical: -6.963752 analytic: -6.963752, relative error: 5.920761e-11
numerical: 4.056253 analytic: 4.056253, relative error: 5.278267e-12
numerical: -3.111351 analytic: -3.111351, relative error: 1.714545e-10
numerical: 12.732826 analytic: 12.732826, relative error: 3.920699e-11
numerical: -4.949063 analytic: -4.949063, relative error: 7.253134e-11
numerical: -3.586712 analytic: -3.586712, relative error: 2.907606e-11
numerical: 10.951367 analytic: 10.951367, relative error: 3.988842e-11
numerical: -29.997081 analytic: -29.997081, relative error: 7.208802e-12
numerical: 0.023334 analytic: 0.023334, relative error: 3.560866e-09
numerical: -3.237983 analytic: -3.237983, relative error: 5.787310e-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*

YourAnswer : Indeed this is possible. Recall the SVM loss function: max(0, x), where x is is the difference between the scores of incorrect classes and correct class plus delta. If x > 0, we incur a loss, else if x < 0, we clamp/threshold the output to 0. A problem with the max function arises when we try to calculate the gradients at a certain value of x where the analytic and numerical gradients mismatch. For instance, the gradient of the SVM loss function is undefined at the hinge, i.e., at x = 0. Generally, when we have max(x, y), at x = y the gradient is undefined. These non-differentiable parts of the function are called "kinks" and they lead to failed gradchecks.

Kinks are not a cause for concern since we can still do gradient descent because the gradients everywhere else apart from the hinge in case of SVM are valid. In practice, it is very rare to actually have your loss be at this precise point in the function where you can't compute the gradient. However, if that happens, it is safe to just skip that gradient update step when doing gradient descent.

Some examples where gradcheck could fail:

- Modulus/absolute function: |x| at x = 0.
- ReLU: max(0, z) at z = 0.
- Simple example: max(0, x) when xis a very small value, say -1e-5. The analytical gradient would be 0 as the max function outputs 0. However, there can be a gradient mismatch if the numerical gradient can yield a different number if h is greater than x, say h = 1e-4.

Increasing the margin (delta), would increase the chances of the class scores being higher (and thus positive) which would mean max(0, x) would output a positive number more often (i.e., the function would tend further away from 0) and thus reduce the frequency of a dimension mismatch during gradcheck which occurs at x = 0.

[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: 8.974389e+00 computed in 0.132339s
Vectorized loss: 8.974389e+00 computed in 0.009878s
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 0.143622s
Vectorized loss and gradient: computed in 0.008859s
```

```
difference: 0.000000
```

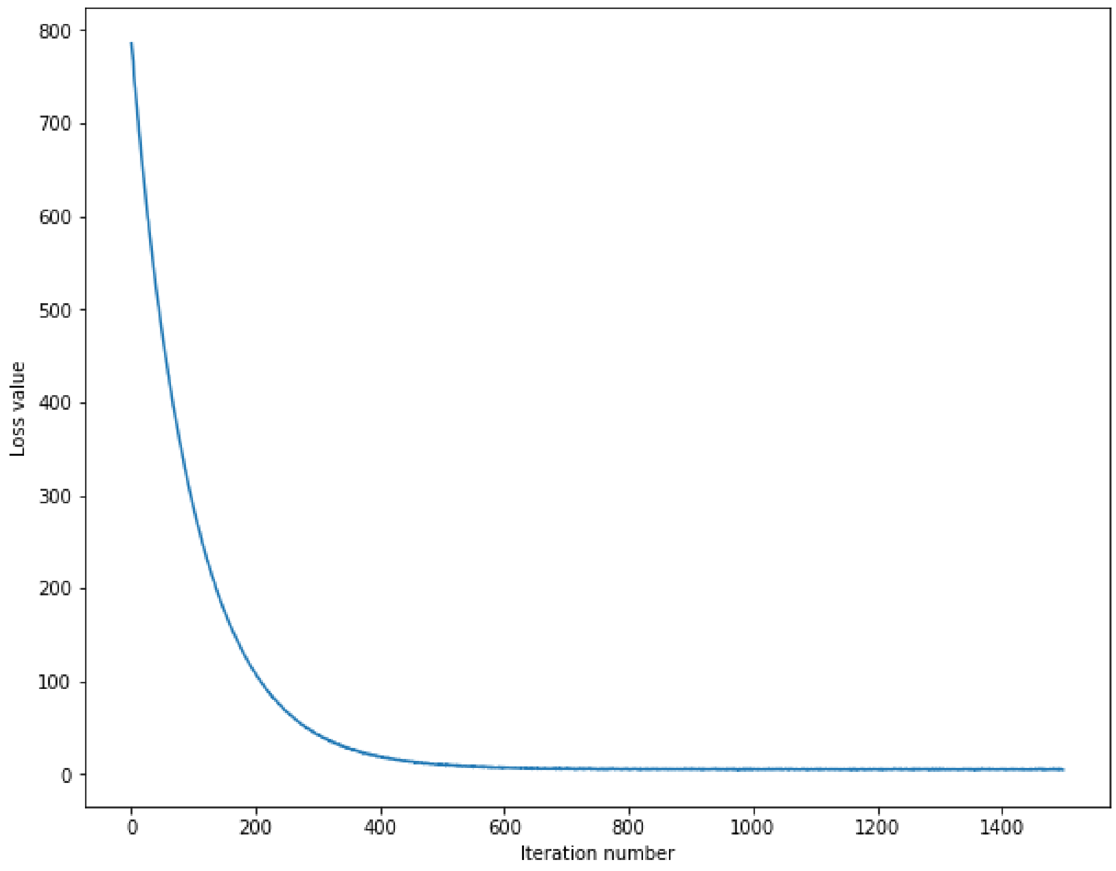
1.2.1 Stochastic Gradient Descent

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 785.729232
iteration 100 / 1500: loss 286.511455
iteration 200 / 1500: loss 107.219561
iteration 300 / 1500: loss 42.420080
iteration 400 / 1500: loss 19.004408
iteration 500 / 1500: loss 10.192130
iteration 600 / 1500: loss 7.473540
iteration 700 / 1500: loss 5.797509
iteration 800 / 1500: loss 5.250400
iteration 900 / 1500: loss 6.217534
iteration 1000 / 1500: loss 5.240777
iteration 1100 / 1500: loss 5.145766
iteration 1200 / 1500: loss 5.002017
iteration 1300 / 1500: loss 6.080431
iteration 1400 / 1500: loss 5.419758
That took 5.676250s
```

[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: 0.371980
validation accuracy: 0.382000
```

[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.                                      ␣
 ↪#
################################################################################

# hyperparamters values to do grid search over
learning_rates = [1e-8, 2e-7, 1e-7, 3e-5] #, 1e-3, 3e-3, 1e-2, 3e-2, 1e-1,␣
 ↪3e-1, 1e1, 3e1, 5e1]
regularization_strengths = [0.5e4, 1e4, 2e4, 2.5e4, 3e4, 3.5e4, 4e4]#, 4.5e4,␣
 ↪5e4, 5.5e4, 6e4]

# permute over learning_rates and regularization_strengths
grid_search = [(lr, reg) for lr in learning_rates \
               for reg in regularization_strengths]

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

```
for config_num, config in enumerate(grid_search):
    print("Hyperparam config #{} of #{}".format(config_num+1, len(grid_search)))
    print("Hyperparam config: {}".format(config))
    lr, reg = config

    svm = LinearSVM()

    # train a linear SVM on the training set
    loss_hist = svm.train(X_train, y_train, learning_rate=lr,
                          reg=reg, num_iters=1500, verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)

    # compute the accuracy on the training and validation sets
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(lr, reg)] = (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the LinearSVM object
    if current_y_val_accuracy > best_val:
        best_val = current_y_val_accuracy
        best_svm = svm

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

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

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

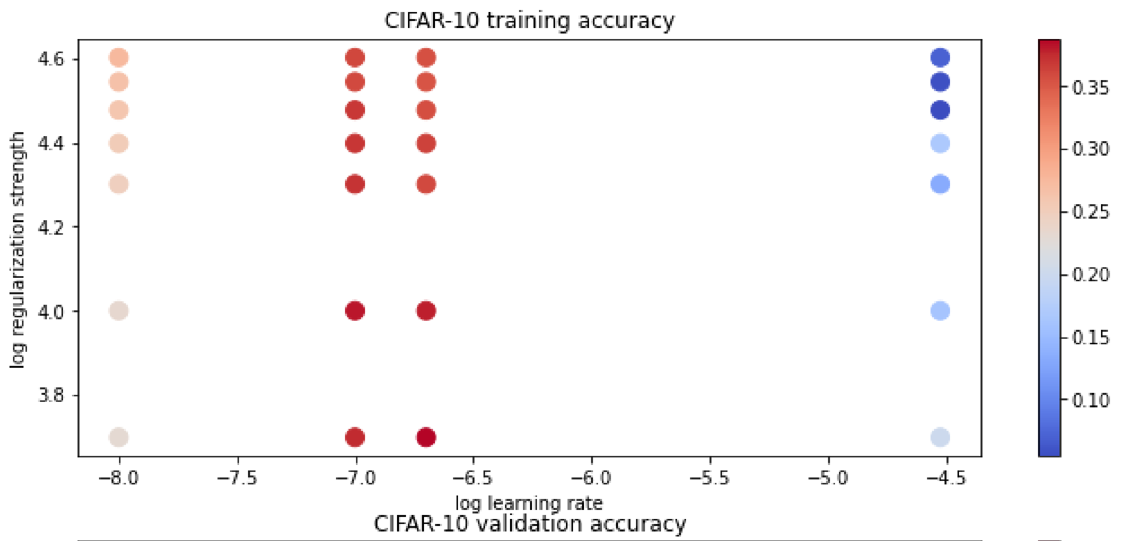
```
Hyperparam config #1 of #28
Hyperparam config: (1e-08, 5000.0)
Hyperparam config #2 of #28
Hyperparam config: (1e-08, 10000.0)
Hyperparam config #3 of #28
Hyperparam config: (1e-08, 20000.0)
Hyperparam config #4 of #28
Hyperparam config: (1e-08, 25000.0)
Hyperparam config #5 of #28
Hyperparam config: (1e-08, 30000.0)
Hyperparam config #6 of #28
Hyperparam config: (1e-08, 35000.0)
Hyperparam config #7 of #28
Hyperparam config: (1e-08, 40000.0)
Hyperparam config #8 of #28
Hyperparam config: (2e-07, 5000.0)
Hyperparam config #9 of #28
Hyperparam config: (2e-07, 10000.0)
Hyperparam config #10 of #28
Hyperparam config: (2e-07, 20000.0)
Hyperparam config #11 of #28
Hyperparam config: (2e-07, 25000.0)
Hyperparam config #12 of #28
Hyperparam config: (2e-07, 30000.0)
Hyperparam config #13 of #28
Hyperparam config: (2e-07, 35000.0)
Hyperparam config #14 of #28
Hyperparam config: (2e-07, 40000.0)
Hyperparam config #15 of #28
Hyperparam config: (1e-07, 5000.0)
Hyperparam config #16 of #28
Hyperparam config: (1e-07, 10000.0)
Hyperparam config #17 of #28
Hyperparam config: (1e-07, 20000.0)
Hyperparam config #18 of #28
Hyperparam config: (1e-07, 25000.0)
Hyperparam config #19 of #28
Hyperparam config: (1e-07, 30000.0)
Hyperparam config #20 of #28
Hyperparam config: (1e-07, 35000.0)
Hyperparam config #21 of #28
Hyperparam config: (1e-07, 40000.0)
Hyperparam config #22 of #28
Hyperparam config: (3e-05, 5000.0)
Hyperparam config #23 of #28
Hyperparam config: (3e-05, 10000.0)
Hyperparam config #24 of #28
Hyperparam config: (3e-05, 20000.0)
Hyperparam config #25 of #28
Hyperparam config: (3e-05, 25000.0)
Hyperparam config #26 of #28
Hyperparam config: (3e-05, 30000.0)
Hyperparam config #27 of #28
Hyperparam config: (3e-05, 35000.0)
Hyperparam config #28 of #28
Hyperparam config: (3e-05, 40000.0)
```

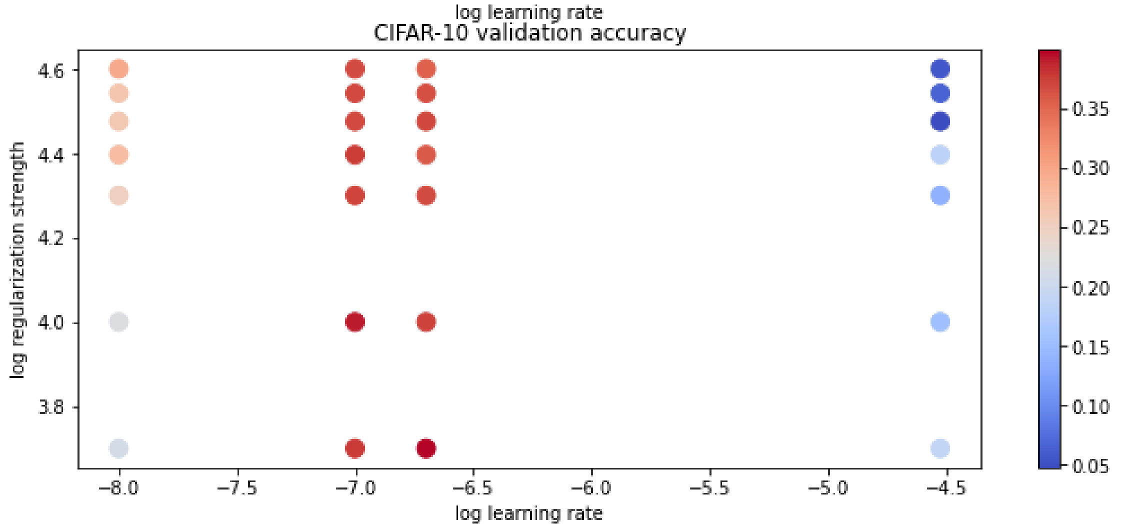
```
/content/cs231n/classifiers/linear_svm.py:131: RuntimeWarning: overflow
encountered in double_scalars
  loss += reg * np.sum(W * W)
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/cs231n/classifiers/linear_svm.py:131: RuntimeWarning: overflow
encountered in multiply
  loss += reg * np.sum(W * W)
```

```
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.230653 val accuracy: 0.211000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.233571 val accuracy: 0.220000
lr 1.000000e-08 reg 2.000000e+04 train accuracy: 0.248020 val accuracy: 0.252000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.252939 val accuracy: 0.276000
lr 1.000000e-08 reg 3.000000e+04 train accuracy: 0.261184 val accuracy: 0.263000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.265857 val accuracy: 0.265000
lr 1.000000e-08 reg 4.000000e+04 train accuracy: 0.274898 val accuracy: 0.297000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.375531 val accuracy: 0.378000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.382694 val accuracy: 0.392000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.371714 val accuracy: 0.372000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.370633 val accuracy: 0.376000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.369653 val accuracy: 0.370000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.359673 val accuracy: 0.369000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.361184 val accuracy: 0.368000
lr 2.000000e-07 reg 5.000000e+03 train accuracy: 0.387939 val accuracy: 0.399000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.379367 val accuracy: 0.373000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.360245 val accuracy: 0.368000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.363184 val accuracy: 0.359000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.357204 val accuracy: 0.371000
lr 2.000000e-07 reg 3.500000e+04 train accuracy: 0.352939 val accuracy: 0.365000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.355959 val accuracy: 0.354000
lr 3.000000e-05 reg 5.000000e+03 train accuracy: 0.200327 val accuracy: 0.191000
lr 3.000000e-05 reg 1.000000e+04 train accuracy: 0.162122 val accuracy: 0.154000
lr 3.000000e-05 reg 2.000000e+04 train accuracy: 0.133245 val accuracy: 0.138000
lr 3.000000e-05 reg 2.500000e+04 train accuracy: 0.167878 val accuracy: 0.185000
lr 3.000000e-05 reg 3.000000e+04 train accuracy: 0.054184 val accuracy: 0.048000
lr 3.000000e-05 reg 3.500000e+04 train accuracy: 0.057510 val accuracy: 0.067000
lr 3.000000e-05 reg 4.000000e+04 train accuracy: 0.071082 val accuracy: 0.057000
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()
```



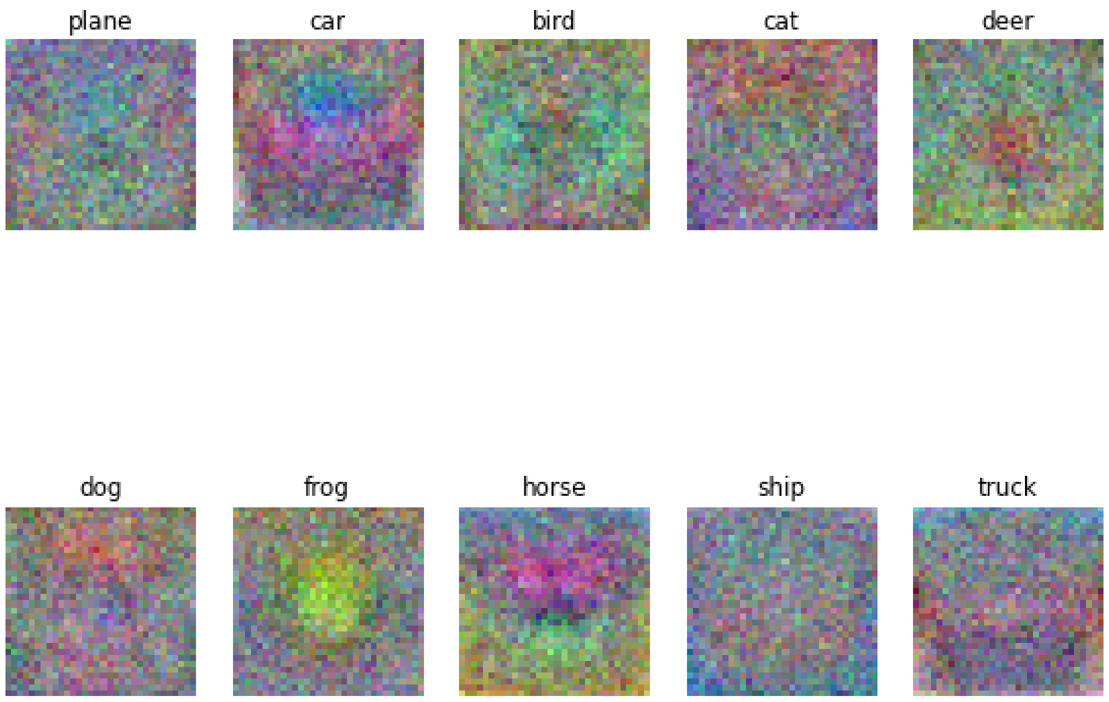


[0]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.369000
```

[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']
```

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

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



Inline question 2

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

Your Answer : The visualized SVM weights represent templates for each class that have been learned from the data. Each of them essentially describe the "essential construction" of the training images that belong to a particular class. For instance, the weights of the class "horse" look like a horse with two heads because the dataset likely has images of horses with some of them looking left and others looking right. With k-NN, we compare a test image with all of the training examples using an appropriate distance measure (say L1 or L2) in order to predict the class of a particular test sample -- however, with SVM, we compare the test image with the templates of each class by using the inner product.

---

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()))
```

softmax

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
--2020-04-19 08:28:16--  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  73.7MB/s    in 2.2s

2020-04-19 08:28:19 (73.7 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 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]:
```python
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
```

[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: 2.344473
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 we are calculating our loss based on random weights (i.e., we haven't started the "learning" process yet), we expect that the initial loss has to be close to -log(0.1) because initially all the classes are equally likely to be chosen. Since CIFAR-10 consists of samples which belong to one of ten classes, the probability of the correct class will be 1/10 = 0.1. The softmax loss is the negative log probability of the correct class, therefore it is -log(0.1).

[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: 1.727196 analytic: 1.727196, relative error: 3.603507e-08
numerical: 0.841981 analytic: 0.841981, relative error: 1.250982e-08
numerical: 0.825645 analytic: 0.825645, relative error: 4.918848e-08
numerical: -1.931518 analytic: -1.931518, relative error: 1.567088e-08
numerical: -3.551808 analytic: -3.551808, relative error: 1.936232e-08
numerical: 0.699782 analytic: 0.699782, relative error: 2.169304e-08
numerical: -1.471077 analytic: -1.471077, relative error: 3.225648e-08
numerical: 3.018008 analytic: 3.018008, relative error: 2.274195e-08
numerical: 1.051361 analytic: 1.051361, relative error: 7.544453e-09
numerical: 2.589550 analytic: 2.589550, relative error: 3.582034e-08
numerical: 2.075771 analytic: 2.075771, relative error: 1.766383e-09
numerical: -0.280870 analytic: -0.280870, relative error: 5.472163e-08
numerical: -0.147655 analytic: -0.147655, relative error: 4.676862e-07
numerical: -0.335186 analytic: -0.335186, relative error: 1.519096e-07
numerical: 0.731971 analytic: 0.731971, relative error: 2.726906e-08
numerical: -1.253421 analytic: -1.253421, relative error: 2.951241e-08
numerical: 0.530505 analytic: 0.530505, relative error: 4.110492e-08
numerical: -4.528923 analytic: -4.528924, relative error: 1.315978e-08
```

```
numerical: -5.601057 analytic: -5.601058, relative error: 5.149598e-09
numerical: -2.693165 analytic: -2.693165, relative error: 1.669678e-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: 2.344473e+00 computed in 0.082723s
vectorized loss: 2.344473e+00 computed in 0.013775s
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 = [1e-8, 2e-7, 1e-7, 2e-6, 3e-5] #, 1e-3, 3e-3, 1e-2, 3e-2,␣
↪1e-1, 3e-1, 1e1, 3e1, 5e1]
regularization_strengths = [0.5e3, 1e3, 0.5e4, 1e4, 2e4, 2.5e4, 3e4, 3.5e4,␣
↪4e4]#, 4.5e4, 5e4, 5.5e4, 6e4]

# permute over learning_rates and regularization_strengths
grid_search = [(learning_rate, regularization_strength) for learning_rate in␣
↪learning_rates \
               for regularization_strength in regularization_strengths]

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

for learning_rate, regularization_strength in grid_search:

    softmax = Softmax()

    # train a linear SVM on the training set
    loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate,
                              reg=regularization_strength, num_iters=1500,␣
↪verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)

    # compute the accuracy on the training and validation sets
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(learning_rate, regularization_strength)] = \
    (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the LinearSVM object
    if current_y_val_accuracy > best_val:
        best_val = current_y_val_accuracy
        best_softmax = softmax
```

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

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

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

```
/content/cs231n/classifiers/softmax.py:102: RuntimeWarning: divide by zero
encountered in log
  loss = -np.log(probs[np.arange(num_train), y])
/content/cs231n/classifiers/softmax.py:123: RuntimeWarning: overflow encountered
in double_scalars
  loss += reg * np.sum(W * W)
/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/softmax.py:123: RuntimeWarning: overflow encountered
in multiply
  loss += reg * np.sum(W * W)
```

```
lr 1.000000e-08 reg 5.000000e+02 train accuracy: 0.162204 val accuracy: 0.158000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.158143 val accuracy: 0.156000
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.167735 val accuracy: 0.147000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.141306 val accuracy: 0.151000
lr 1.000000e-08 reg 2.000000e+04 train accuracy: 0.146122 val accuracy: 0.156000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.171306 val accuracy: 0.180000
lr 1.000000e-08 reg 3.000000e+04 train accuracy: 0.185347 val accuracy: 0.203000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.188347 val accuracy: 0.201000
lr 1.000000e-08 reg 4.000000e+04 train accuracy: 0.191449 val accuracy: 0.203000
lr 1.000000e-07 reg 5.000000e+02 train accuracy: 0.253469 val accuracy: 0.277000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.263735 val accuracy: 0.245000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.331755 val accuracy: 0.357000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.355571 val accuracy: 0.362000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.339102 val accuracy: 0.354000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329408 val accuracy: 0.339000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.324143 val accuracy: 0.339000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.318755 val accuracy: 0.334000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.312796 val accuracy: 0.326000
lr 2.000000e-07 reg 5.000000e+02 train accuracy: 0.297878 val accuracy: 0.300000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.318612 val accuracy: 0.315000
lr 2.000000e-07 reg 5.000000e+03 train accuracy: 0.373735 val accuracy: 0.382000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.357837 val accuracy: 0.377000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.340918 val accuracy: 0.357000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.329571 val accuracy: 0.335000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.331122 val accuracy: 0.350000
lr 2.000000e-07 reg 3.500000e+04 train accuracy: 0.309878 val accuracy: 0.328000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.310612 val accuracy: 0.322000
lr 2.000000e-06 reg 5.000000e+02 train accuracy: 0.404673 val accuracy: 0.403000
lr 2.000000e-06 reg 1.000000e+03 train accuracy: 0.393755 val accuracy: 0.395000
lr 2.000000e-06 reg 5.000000e+03 train accuracy: 0.363184 val accuracy: 0.371000
lr 2.000000e-06 reg 1.000000e+04 train accuracy: 0.347694 val accuracy: 0.369000
lr 2.000000e-06 reg 2.000000e+04 train accuracy: 0.305408 val accuracy: 0.313000
lr 2.000000e-06 reg 2.500000e+04 train accuracy: 0.315755 val accuracy: 0.321000
lr 2.000000e-06 reg 3.000000e+04 train accuracy: 0.304429 val accuracy: 0.294000
lr 2.000000e-06 reg 3.500000e+04 train accuracy: 0.298837 val accuracy: 0.296000
lr 2.000000e-06 reg 4.000000e+04 train accuracy: 0.283694 val accuracy: 0.315000
lr 3.000000e-05 reg 5.000000e+02 train accuracy: 0.206490 val accuracy: 0.196000
lr 3.000000e-05 reg 1.000000e+03 train accuracy: 0.229694 val accuracy: 0.239000
lr 3.000000e-05 reg 5.000000e+03 train accuracy: 0.133265 val accuracy: 0.133000
lr 3.000000e-05 reg 1.000000e+04 train accuracy: 0.102735 val accuracy: 0.092000
lr 3.000000e-05 reg 2.000000e+04 train accuracy: 0.078551 val accuracy: 0.080000
lr 3.000000e-05 reg 2.500000e+04 train accuracy: 0.078122 val accuracy: 0.070000
lr 3.000000e-05 reg 3.000000e+04 train accuracy: 0.060796 val accuracy: 0.054000
lr 3.000000e-05 reg 3.500000e+04 train accuracy: 0.102551 val accuracy: 0.122000
lr 3.000000e-05 reg 4.000000e+04 train accuracy: 0.078551 val accuracy: 0.066000
best validation accuracy achieved during cross-validation: 0.403000
```

[0]:
```
# 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: 0.398000
```

Inline Question 2 - *True or False*

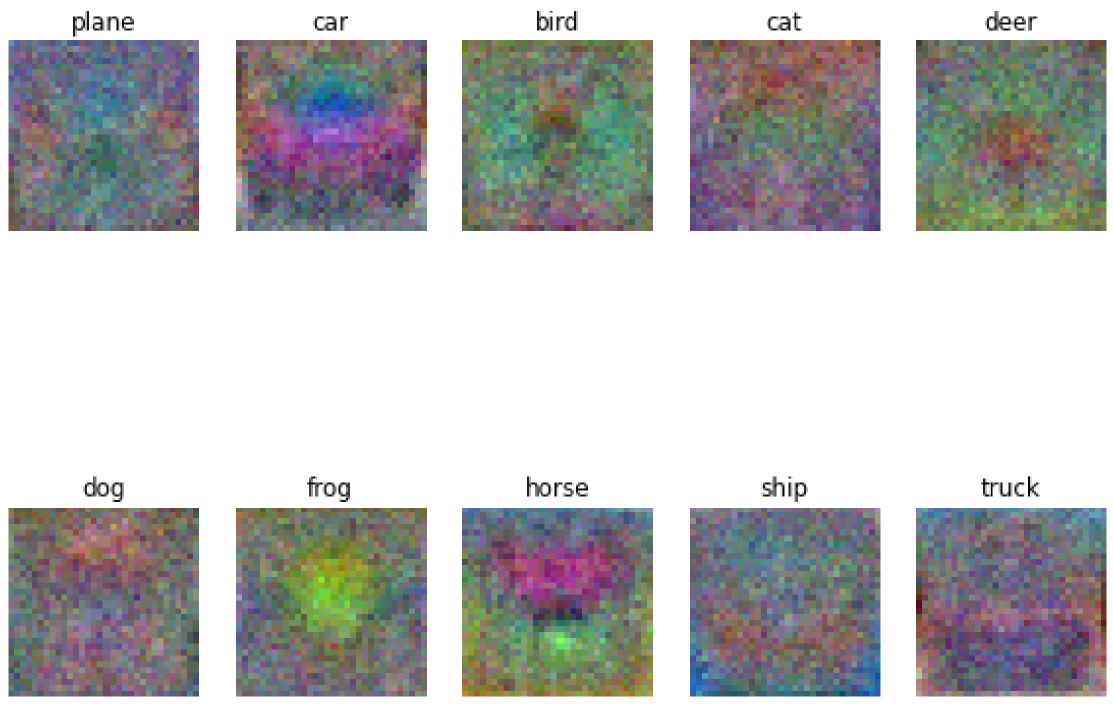
Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your Answer : True.

Your Explanation : As long as the SVM loss of the new datapoint is zero, the SVM loss would be unaffected. This would be the case if its incorrect classes satisfy the margin, i.e., the scores of the incorrect class is below the score of the correct class by atleast delta. Assume the following: (i) the delta for SVM is 1, (ii) the correct class is #0 and, (iii) we add a new datapoint #3 that leads to scores [3,1,2], then the SVM loss for the new datapoint will be 0 (and the overall SVM loss would be unchanged) because it is below the score of the correct class by atleast delta, i.e., max(0, 1 - 3 + 1) + max(0, 2 - 3 + 1) = 0.

Unlike SVM, the Softmax classifier will assume a larger loss because the addition of a new datapoint will ultimately reduce the score of the correct class after normalization which would affect the loss negatively (i.e., lesser the score of the correct class, more the loss). This is because the Softmax classifier considers all the individual scores in the calculation of its loss while the SVM loss does not take into account the individual scores as long as the incorrect classes have scores that are below that of the correct class by atleast delta. Put differently, how far below the scores of the incorrect classes are compared to that of the correct class (as long as they are atleast delta apart) is not a point of consideration for SVM, unlike Softmax.

[0]:
```
# 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]:
```python
import os

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

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

two_layer_net

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 ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-19 08:27:38--  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  74.2MB/s    in 2.2s

2020-04-19 08:27:41 (74.2 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]
```

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

1 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]:
```python
# 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))))
```

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

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

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

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

```
Difference between your loss and correct loss:
1.7985612998927536e-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]:
```python
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

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

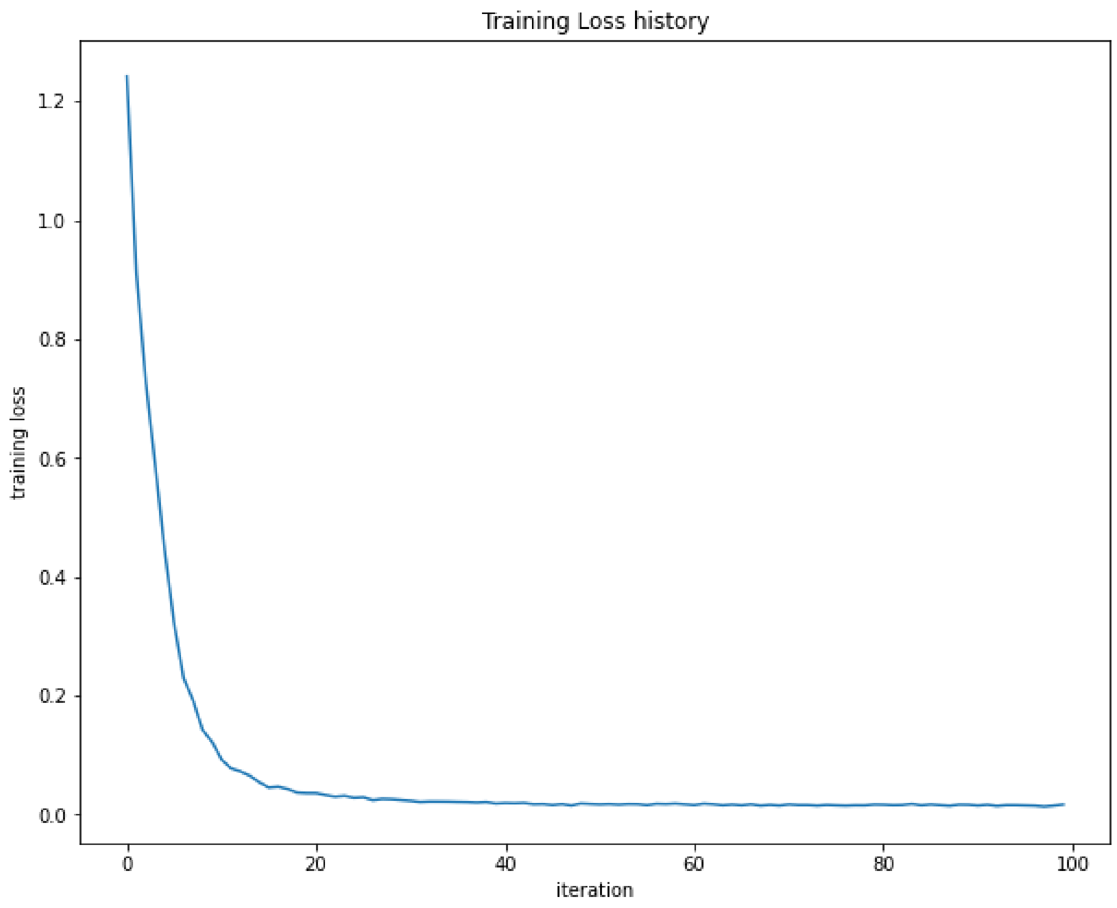
5 Train the network

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

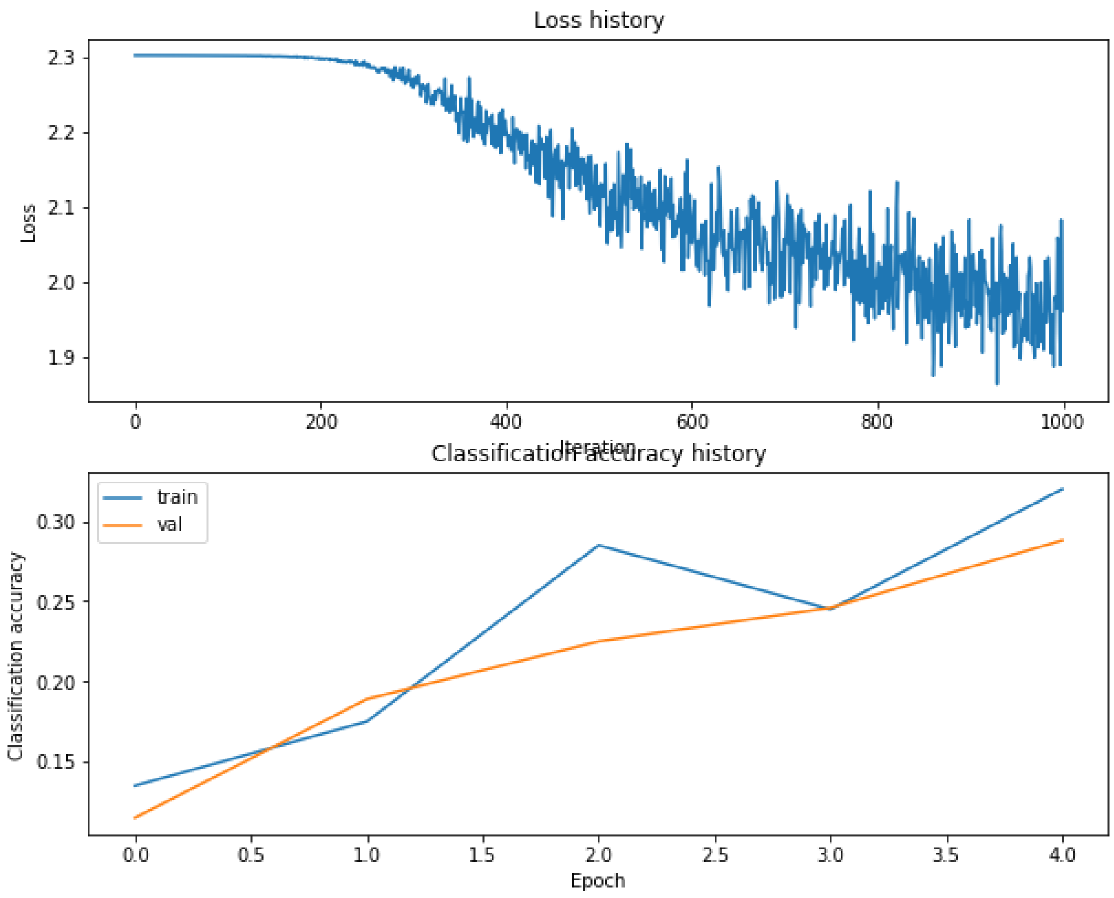
8 Debug the training

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

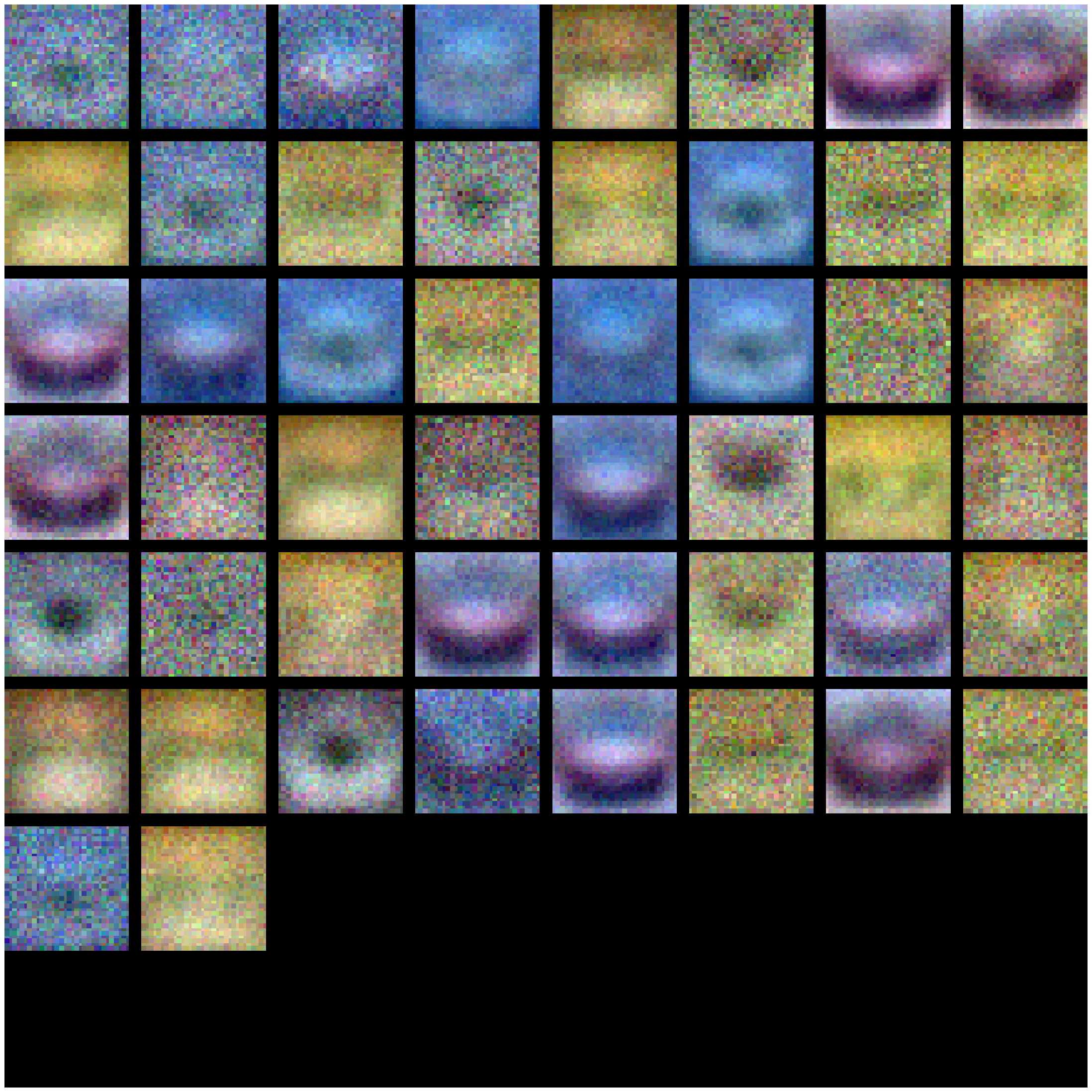
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

[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 (and that they are both relatively low, ~30%), suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

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

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

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

Explain your hyperparameter tuning process below.

$Your Answer :$ We performed random search-based hyperparameter tuning over a range of hyperparameters such as hidden layer size, learning rate, number of training epochs, and regularization strength. Random search is a technique where hyperparameter values are randomzied and combinations of these randomized hyperparameter values are used to find the sweet spot for the model's performance. The design space of hyperparameters can be huge as it grows exponentially with the addition of every hyperpamater, leading to the curse of dimensionality. To that end, we optimize random search by considering randomly-selected pre-defined hyperparameter configurations in the parameter space and evaluate the network's performance at these points. In our implementation, we have parameterized the number of random hyperparameter combinations to look for, to enable deep searches if need be. The other alternative is grid search where every combination of a pre-set list of reasonable hyperparameter values is exhaustively evaluated against the model.

Since grid search grows exponentially in complexity as the number of hyperparameters (i.e., the dimensionality of our search) increases, it isn't the optimal choice for our scenario given the number of hyperparameter values we would need to search over.

Also, in "Random Search for Hyper-Parameter Optimization" by Bergstra and Bengio, the authors show theoretically and empirically that random search is more efficient for hyperparameter optimization than grid search. They show that this is especially true for lower dimensional data since the time taken to find the optimum parameter set is lesser than grid search since it uses less number of iterations.

[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)*****

input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10

# generate random combinations of hyperparameters given arrays of potential␣
↪values
def random_hyperparams_search(lr_values, reg_values, hidden_size_values,␣
↪epoch_values):
    lr = lr_values[np.random.randint(0, len(lr_values))]
    reg = reg_values[np.random.randint(0, len(reg_values))]
    hidden_size = hidden_size_values[np.random.randint(0,␣
↪len(hidden_size_values))]
    epochs = epoch_values[np.random.randint(0, len(epoch_values))]
    return lr, reg, hidden_size, epochs

# generate random combinations of hyperparameters given min-max ranges of␣
↪hyperparams
# using a uniform probability distribution
# usage: lr, reg, hidden_dim = generate_random_hyperparams((-1, 0), (-7, -4),␣
↪(10, 500))
#def generate_random_hyperparams(lr, reg, hidden_size):
#    lr = 10**np.random.uniform(lr[0], lr[1])
#    reg = 10**np.random.uniform(reg[0], reg[1])
#    hidden_size = np.random.randint(hidden_size[0], hidden_size[1])
#    return lr, reg, hidden_size

# number of hypercombinations combinations to look for
num_hyperparam_configs = 10
```

```
# randomly permute over learning_rate, regularization_strength,␣
↪hidden_layer_size and num_training_epochs
grid_search = [random_hyperparams_search([0.001], [0.05, 0.1, 0.15, 0.5], [30,␣
↪50, 80, 100, 150, 200], [1500, 3000])
               for count in range(num_hyperparam_configs)]

# get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()

best_val = -1   # The highest validation accuracy that we have seen so far.
results = {}

for config_num, config in enumerate(grid_search):
    print("Hyperparam config #{} of #{}".format(config_num+1, len(grid_search)))
    print("Hyperparam config: {}".format(config))

    lr, reg, hidden_size, epochs = config

    net = TwoLayerNet(input_size, hidden_size, num_classes)

    # train a 2-layer neural net on the training set
    loss_hist = net.train(X_train, y_train, X_val, y_val,
                          learning_rate=lr, reg=reg, num_iters=epochs,
                          batch_size=200, verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = net.predict(X_train)
    y_val_pred = net.predict(X_val)

    # compute the accuracy on the training and validation sets
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(lr, reg, hidden_size, epochs)] = \
            (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the TwoLayerNet object
    if current_y_val_accuracy > best_val:
        best_val = current_y_val_accuracy
        best_net = net

# Print out results.
for lr, reg, hidden_size, num_training_epochs in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg, hidden_size,␣
↪num_training_epochs)]
```

```
    print('lr %e reg %e hidden_size %e num_training_epochs %e train accuracy:␣
↪%f val accuracy: %f' % (
            lr, reg, hidden_size, num_training_epochs, train_accuracy,␣
↪val_accuracy))

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

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

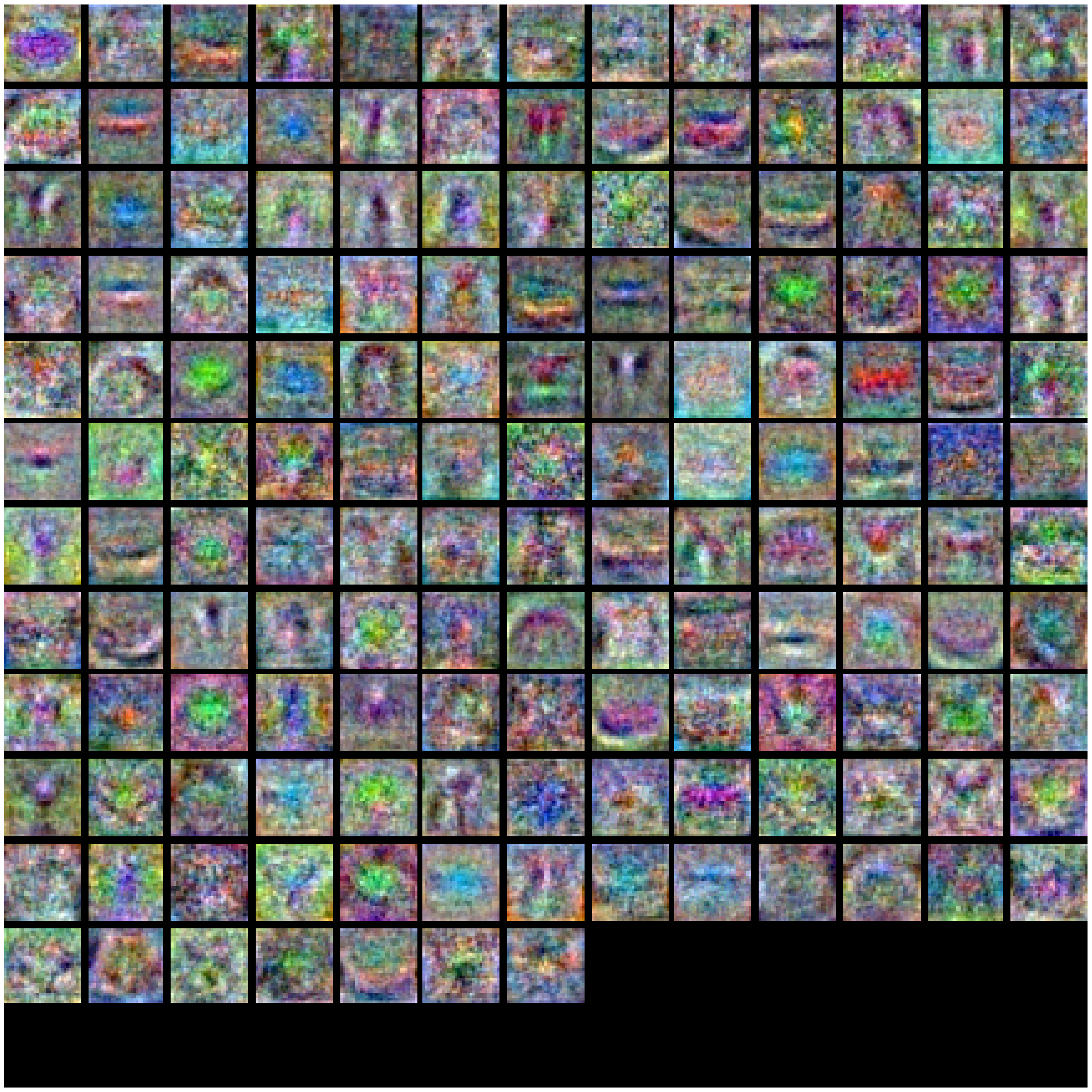
```
Hyperparam config #1 of #10
Hyperparam config: (0.001, 0.05, 50, 3000)
Hyperparam config #2 of #10
Hyperparam config: (0.001, 0.5, 80, 1500)
Hyperparam config #3 of #10
Hyperparam config: (0.001, 0.5, 150, 3000)
Hyperparam config #4 of #10
Hyperparam config: (0.001, 0.1, 50, 1500)
Hyperparam config #5 of #10
Hyperparam config: (0.001, 0.1, 150, 1500)
Hyperparam config #6 of #10
Hyperparam config: (0.001, 0.1, 80, 3000)
Hyperparam config #7 of #10
Hyperparam config: (0.001, 0.1, 200, 3000)
Hyperparam config #8 of #10
Hyperparam config: (0.001, 0.1, 80, 1500)
Hyperparam config #9 of #10
Hyperparam config: (0.001, 0.15, 100, 1500)
Hyperparam config #10 of #10
Hyperparam config: (0.001, 0.1, 200, 1500)
lr 1.000000e-03 reg 5.000000e-02 hidden_size 5.000000e+01 num_training_epochs
3.000000e+03 train accuracy: 0.554571 val accuracy: 0.499000
lr 1.000000e-03 reg 1.000000e-01 hidden_size 5.000000e+01 num_training_epochs
1.500000e+03 train accuracy: 0.519184 val accuracy: 0.496000
lr 1.000000e-03 reg 1.000000e-01 hidden_size 8.000000e+01 num_training_epochs
1.500000e+03 train accuracy: 0.524918 val accuracy: 0.495000
lr 1.000000e-03 reg 1.000000e-01 hidden_size 8.000000e+01 num_training_epochs
3.000000e+03 train accuracy: 0.581061 val accuracy: 0.542000
lr 1.000000e-03 reg 1.000000e-01 hidden_size 1.500000e+02 num_training_epochs
1.500000e+03 train accuracy: 0.540490 val accuracy: 0.516000
lr 1.000000e-03 reg 1.000000e-01 hidden_size 2.000000e+02 num_training_epochs
1.500000e+03 train accuracy: 0.548143 val accuracy: 0.490000
lr 1.000000e-03 reg 1.000000e-01 hidden_size 2.000000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.615735 val accuracy: 0.526000
lr 1.000000e-03 reg 1.500000e-01 hidden_size 1.000000e+02 num_training_epochs
1.500000e+03 train accuracy: 0.526327 val accuracy: 0.483000
lr 1.000000e-03 reg 5.000000e-01 hidden_size 8.000000e+01 num_training_epochs
```

```
1.500000e+03 train accuracy: 0.512633 val accuracy: 0.492000
lr 1.000000e-03 reg 5.000000e-01 hidden_size 1.500000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.555571 val accuracy: 0.518000
best validation accuracy achieved during cross-validation: 0.542000
```

[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:  0.525
```

[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]:
```python
# 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.

$\mathit{Your Answer}:$ 1, 3

$\mathit{Your Explanation}:$

The testing accuracy being lower than the training accuracy (i.e., lack of generalization) is likely due to the model overfitting the training data.

1. True. Training on a larger dataset helps prevent overfitting. For instance in the CIFAR dataset, adding more samples of each class will exapnd the diversity of the intra-class variations that the model has been exposed to, and thus allow the model to generalize better. An important caveat here is that if the additional data is noisy or similar to what the model has already seen before, this will not help in the model's learning process.
2. False. The model's training accuracy being much higher than the testing accuracy indicates that the model is inherently complex enough to learn the features within the data. Adding more hidden units allows the model to learn more complex functions. Had both the training accuracy and training accuracy been low, this would have indicated that the model is not complex enough to learn nuances in the data in which case, adding more hidden units would have helped.
3. True. Increasing the regularization strength helps avoid overfitting because it reduces the complexity of the model by reducing the dependency of the model on "outlier" features (say, noise in the training data) and tends to distribute the weights across all features as much as possible.

---

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()))
```

features

April 23, 2020

[0]:
```python
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
--2020-04-19 08:01:51--  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  87.8MB/s    in 1.9s

2020-04-19 08:01:53 (87.8 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

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

1 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]:
```python
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]:
```python
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.

[0]:
```
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.

[0]:
```python
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-7, 1e-5, 1e-3]
regularization_strengths = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]

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)*****

# permute over learning_rates and regularization_strengths
grid_search = [(lr, reg) for lr in learning_rates \
               for reg in regularization_strengths]

for config_num, config in enumerate(grid_search):
    print("Hyperparam config #{} of #{}".format(config_num+1, len(grid_search)))
    print("Hyperparam config: {}".format(config))
    lr, reg = config

    svm = LinearSVM()

    # train a linear SVM on the training set
    loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr,
                          reg=reg, num_iters=2000, verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = svm.predict(X_train_feats)
    y_val_pred = svm.predict(X_val_feats)

    # compute the accuracy on the training and validation sets
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(lr, reg)] = (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the LinearSVM object
    if current_y_val_accuracy > best_val:
        best_val = current_y_val_accuracy
        best_svm = svm
```

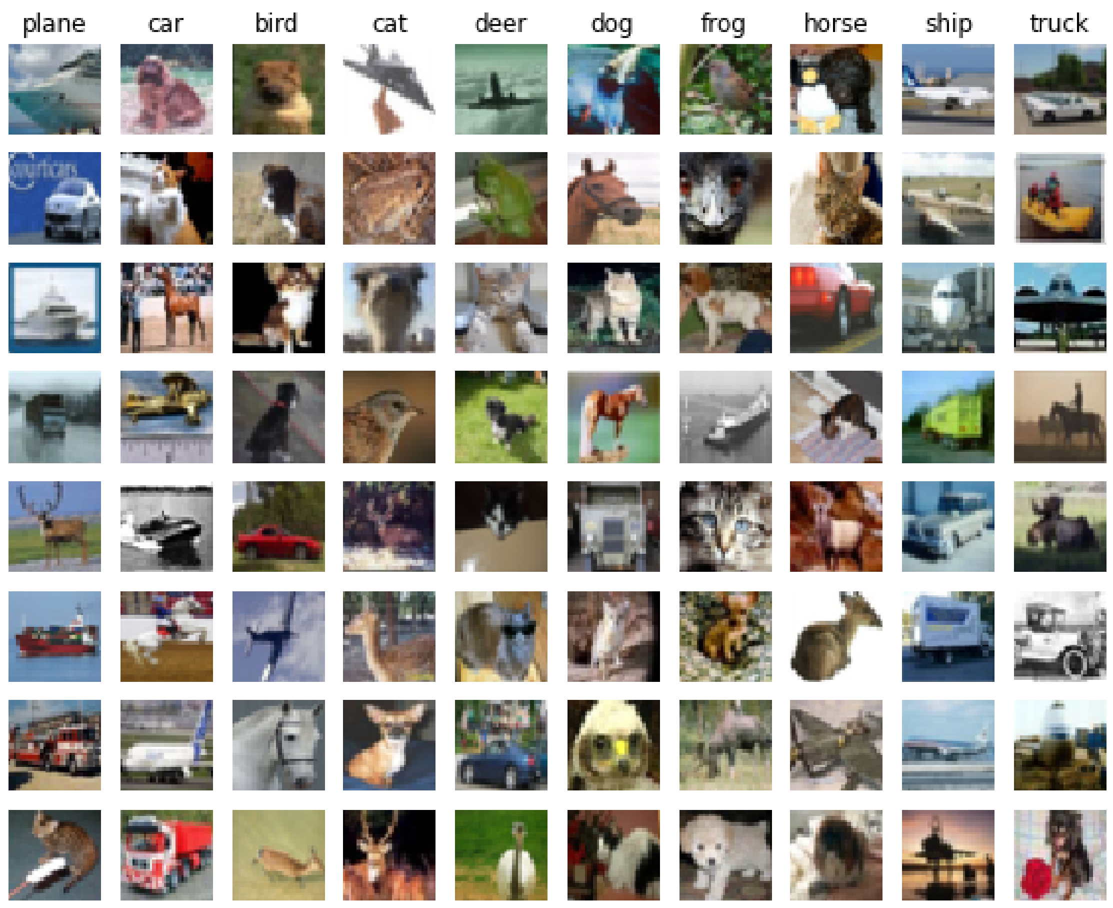
```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

```
Hyperparam config #1 of #36
Hyperparam config: (1e-09, 0.01)
Hyperparam config #2 of #36
Hyperparam config: (1e-09, 0.05)
Hyperparam config #3 of #36
Hyperparam config: (1e-09, 0.1)
Hyperparam config #4 of #36
Hyperparam config: (1e-09, 0.2)
Hyperparam config #5 of #36
Hyperparam config: (1e-09, 0.3)
Hyperparam config #6 of #36
Hyperparam config: (1e-09, 0.4)
Hyperparam config #7 of #36
Hyperparam config: (1e-09, 0.5)
Hyperparam config #8 of #36
Hyperparam config: (1e-09, 0.6)
Hyperparam config #9 of #36
Hyperparam config: (1e-09, 0.7)
Hyperparam config #10 of #36
Hyperparam config: (1e-07, 0.01)
Hyperparam config #11 of #36
Hyperparam config: (1e-07, 0.05)
Hyperparam config #12 of #36
Hyperparam config: (1e-07, 0.1)
Hyperparam config #13 of #36
Hyperparam config: (1e-07, 0.2)
Hyperparam config #14 of #36
Hyperparam config: (1e-07, 0.3)
Hyperparam config #15 of #36
Hyperparam config: (1e-07, 0.4)
Hyperparam config #16 of #36
Hyperparam config: (1e-07, 0.5)
Hyperparam config #17 of #36
Hyperparam config: (1e-07, 0.6)
Hyperparam config #18 of #36
Hyperparam config: (1e-07, 0.7)
Hyperparam config #19 of #36
Hyperparam config: (1e-05, 0.01)
Hyperparam config #20 of #36
Hyperparam config: (1e-05, 0.05)
Hyperparam config #21 of #36
Hyperparam config: (1e-05, 0.1)
Hyperparam config #22 of #36
Hyperparam config: (1e-05, 0.2)
Hyperparam config #23 of #36
Hyperparam config: (1e-05, 0.3)
Hyperparam config #24 of #36
Hyperparam config: (1e-05, 0.4)
Hyperparam config #25 of #36
Hyperparam config: (1e-05, 0.5)
Hyperparam config #26 of #36
Hyperparam config: (1e-05, 0.6)
Hyperparam config #27 of #36
Hyperparam config: (1e-05, 0.7)
Hyperparam config #28 of #36
Hyperparam config: (0.001, 0.01)
Hyperparam config #29 of #36
Hyperparam config: (0.001, 0.05)
Hyperparam config #30 of #36
Hyperparam config: (0.001, 0.1)
Hyperparam config #31 of #36
Hyperparam config: (0.001, 0.2)
Hyperparam config #32 of #36
Hyperparam config: (0.001, 0.3)
Hyperparam config #33 of #36
Hyperparam config: (0.001, 0.4)
Hyperparam config #34 of #36
Hyperparam config: (0.001, 0.5)
Hyperparam config #35 of #36
Hyperparam config: (0.001, 0.6)
Hyperparam config #36 of #36
Hyperparam config: (0.001, 0.7)
lr 1.000000e-09 reg 1.000000e-02 train accuracy: 0.100204 val accuracy: 0.092000
lr 1.000000e-09 reg 5.000000e-02 train accuracy: 0.093837 val accuracy: 0.100000
lr 1.000000e-09 reg 1.000000e-01 train accuracy: 0.092898 val accuracy: 0.083000
lr 1.000000e-09 reg 2.000000e-01 train accuracy: 0.107980 val accuracy: 0.113000
lr 1.000000e-09 reg 3.000000e-01 train accuracy: 0.118163 val accuracy: 0.102000
lr 1.000000e-09 reg 4.000000e-01 train accuracy: 0.123122 val accuracy: 0.110000
lr 1.000000e-09 reg 5.000000e-01 train accuracy: 0.086939 val accuracy: 0.081000
lr 1.000000e-09 reg 6.000000e-01 train accuracy: 0.100143 val accuracy: 0.100000
lr 1.000000e-09 reg 7.000000e-01 train accuracy: 0.098878 val accuracy: 0.080000
lr 1.000000e-07 reg 1.000000e-02 train accuracy: 0.134469 val accuracy: 0.123000
lr 1.000000e-07 reg 5.000000e-02 train accuracy: 0.119367 val accuracy: 0.118000
lr 1.000000e-07 reg 1.000000e-01 train accuracy: 0.119857 val accuracy: 0.125000
lr 1.000000e-07 reg 2.000000e-01 train accuracy: 0.110898 val accuracy: 0.114000
lr 1.000000e-07 reg 3.000000e-01 train accuracy: 0.119837 val accuracy: 0.132000
lr 1.000000e-07 reg 4.000000e-01 train accuracy: 0.137878 val accuracy: 0.147000
lr 1.000000e-07 reg 5.000000e-01 train accuracy: 0.114408 val accuracy: 0.122000
lr 1.000000e-07 reg 6.000000e-01 train accuracy: 0.135714 val accuracy: 0.136000
lr 1.000000e-07 reg 7.000000e-01 train accuracy: 0.148367 val accuracy: 0.161000
lr 1.000000e-05 reg 1.000000e-02 train accuracy: 0.410347 val accuracy: 0.411000
lr 1.000000e-05 reg 5.000000e-02 train accuracy: 0.411837 val accuracy: 0.406000
lr 1.000000e-05 reg 1.000000e-01 train accuracy: 0.413551 val accuracy: 0.403000
lr 1.000000e-05 reg 2.000000e-01 train accuracy: 0.408918 val accuracy: 0.419000
lr 1.000000e-05 reg 3.000000e-01 train accuracy: 0.416857 val accuracy: 0.416000
lr 1.000000e-05 reg 4.000000e-01 train accuracy: 0.411653 val accuracy: 0.424000
lr 1.000000e-05 reg 5.000000e-01 train accuracy: 0.411388 val accuracy: 0.413000
lr 1.000000e-05 reg 6.000000e-01 train accuracy: 0.413265 val accuracy: 0.410000
lr 1.000000e-05 reg 7.000000e-01 train accuracy: 0.413265 val accuracy: 0.416000
lr 1.000000e-03 reg 1.000000e-02 train accuracy: 0.503918 val accuracy: 0.495000
lr 1.000000e-03 reg 5.000000e-02 train accuracy: 0.503959 val accuracy: 0.491000
lr 1.000000e-03 reg 1.000000e-01 train accuracy: 0.503592 val accuracy: 0.491000
lr 1.000000e-03 reg 2.000000e-01 train accuracy: 0.498633 val accuracy: 0.483000
lr 1.000000e-03 reg 3.000000e-01 train accuracy: 0.498102 val accuracy: 0.490000
lr 1.000000e-03 reg 4.000000e-01 train accuracy: 0.495469 val accuracy: 0.479000
lr 1.000000e-03 reg 5.000000e-01 train accuracy: 0.493102 val accuracy: 0.476000
lr 1.000000e-03 reg 6.000000e-01 train accuracy: 0.489959 val accuracy: 0.473000
lr 1.000000e-03 reg 7.000000e-01 train accuracy: 0.489755 val accuracy: 0.483000
best validation accuracy achieved during cross-validation: 0.495000
```

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

```
0.475
```

[0]:
```python
# 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. For instance, an occasional sample of a cat, dog or even a car features in the bird class and vice versa. Cats, dogs, birds and other animals have similar "appearance" in that they have similar textures due to common facial features (eyes, ears, nose, mouth) and in some cases, quadruped legs, which HOG captures; they also have similar color schemes, which the color histogram represents. Similarly, horses and deers are mixed-up owing to similar textures and color features. Cars and trucks are mixed-up based on a similar argument. There are several class mismatches due to the background similarites as well - for instance, consider the plane class where the misclassified examples have similarities in the background because the sky and sea colors are similar, which is why they are commonly mixed up with ship images.

In conclusion, the combination of HOG and color histogram feature vectors are not enough to distinguish between these classes with impeccable accuracy. The HOG descriptor is useful because it takes into account textures within an image, but HOG does not take into account types of intra-class variance such as rotation, scaling, translation, illumination, posture deformation etc.

Scale-invariant feature transform (SIFT) by Lowe can help improve the accuracy of our model in such scenarios. Color histogram features help discriminating classes on the basis of color schemes. We can weigh color histograms lower when using them in conjuction with more advanced feature descriptors like HOG or SIFT to reduce the problem of simple color-based class segregation.

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.

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

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

# generate random combinations of hyperparameters given arrays of potential␣
↪values
def random_hyperparams_search(lr_values, reg_values, hidden_size_values,␣
↪epoch_values):
    lr = lr_values[np.random.randint(0, len(lr_values))]
    reg = reg_values[np.random.randint(0, len(reg_values))]
    hidden_size = hidden_size_values[np.random.randint(0,␣
↪len(hidden_size_values))]
    epochs = epoch_values[np.random.randint(0, len(epoch_values))]
    return lr, reg, hidden_size, epochs

# number of hypercombinations combinations to look for
num_hyperparam_configs = 30

# randomly permute over learning_rate, regularization_strength,␣
↪hidden_layer_size and num_training_epochs
grid_search = [random_hyperparams_search([3e-1, 2e-1, 1e-1, 1e-2, 1e-3], [2e-7,␣
↪1e-7, 1e-6, 1e-5], [30, 50, 80, 100, 150, 200], [3000, 5000])
               for count in range(num_hyperparam_configs)]

best_val = -1   # The highest validation accuracy that we have seen so far.
results = {}

for config_num, config in enumerate(grid_search):
    print("Hyperparam config #{} of #{}".format(config_num+1, len(grid_search)))
    print("Hyperparam config: {}".format(config))

    lr, reg, hidden_size, epochs = config

    net = TwoLayerNet(input_dim, hidden_size, num_classes)

    # train a 2-layer neural net on the training set
    loss_hist = net.train(X_train_feats, y_train, X_val_feats, y_val,
                      learning_rate=lr, reg=reg, num_iters=epochs,
                      batch_size=200, verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = net.predict(X_train_feats)
    y_val_pred = net.predict(X_val_feats)

    # compute the accuracy on the training and validation sets
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(lr, reg, hidden_size, epochs)] = \
            (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the TwoLayerNet object
    if current_y_val_accuracy > best_val:
        best_val = current_y_val_accuracy
        best_net = net

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

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

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

```
Hyperparam config #1 of #30
Hyperparam config: (0.001, 1e-06, 80, 5000)
Hyperparam config #2 of #30
Hyperparam config: (0.1, 1e-06, 100, 3000)
Hyperparam config #3 of #30
Hyperparam config: (0.1, 1e-07, 200, 5000)
Hyperparam config #4 of #30
Hyperparam config: (0.01, 2e-07, 200, 5000)
Hyperparam config #5 of #30
Hyperparam config: (0.2, 1e-05, 200, 3000)
Hyperparam config #6 of #30
Hyperparam config: (0.2, 1e-07, 150, 3000)
Hyperparam config #7 of #30
Hyperparam config: (0.01, 1e-05, 150, 3000)
Hyperparam config #8 of #30
Hyperparam config: (0.001, 1e-06, 80, 3000)
Hyperparam config #9 of #30
Hyperparam config: (0.3, 2e-07, 150, 3000)
Hyperparam config #10 of #30
Hyperparam config: (0.001, 1e-06, 80, 5000)
Hyperparam config #11 of #30
Hyperparam config: (0.001, 1e-07, 200, 5000)
Hyperparam config #12 of #30
Hyperparam config: (0.001, 1e-07, 50, 3000)
Hyperparam config #13 of #30
Hyperparam config: (0.001, 2e-07, 100, 5000)
Hyperparam config #14 of #30
Hyperparam config: (0.001, 2e-07, 30, 3000)
Hyperparam config #15 of #30
Hyperparam config: (0.3, 1e-06, 150, 5000)
Hyperparam config #16 of #30
Hyperparam config: (0.3, 1e-05, 100, 5000)
Hyperparam config #17 of #30
Hyperparam config: (0.1, 1e-05, 50, 5000)
Hyperparam config #18 of #30
Hyperparam config: (0.3, 1e-07, 30, 5000)
Hyperparam config #19 of #30
Hyperparam config: (0.01, 2e-07, 100, 3000)
Hyperparam config #20 of #30
Hyperparam config: (0.2, 1e-07, 150, 3000)
Hyperparam config #21 of #30
Hyperparam config: (0.3, 2e-07, 100, 5000)
Hyperparam config #22 of #30
Hyperparam config: (0.2, 1e-06, 50, 5000)
Hyperparam config #23 of #30
Hyperparam config: (0.2, 2e-07, 50, 3000)
Hyperparam config #24 of #30
Hyperparam config: (0.01, 1e-06, 50, 5000)
Hyperparam config #25 of #30
Hyperparam config: (0.1, 2e-07, 100, 5000)
Hyperparam config #26 of #30
Hyperparam config: (0.3, 1e-05, 150, 3000)
Hyperparam config #27 of #30
Hyperparam config: (0.3, 1e-06, 50, 5000)
Hyperparam config #28 of #30
Hyperparam config: (0.1, 1e-07, 100, 3000)
Hyperparam config #29 of #30
Hyperparam config: (0.2, 1e-05, 50, 3000)
Hyperparam config #30 of #30
Hyperparam config: (0.2, 1e-05, 200, 3000)
lr 1.000000e-03 reg 1.000000e-07 hidden_size 5.000000e+01 num_training_epochs
3.000000e+03 train accuracy: 0.099898 val accuracy: 0.105000
lr 1.000000e-03 reg 1.000000e-07 hidden_size 2.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-03 reg 2.000000e-07 hidden_size 3.000000e+01 num_training_epochs
3.000000e+03 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-03 reg 2.000000e-07 hidden_size 1.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.100429 val accuracy: 0.079000
lr 1.000000e-03 reg 1.000000e-06 hidden_size 8.000000e+01 num_training_epochs
```

```
3.000000e+03 train accuracy: 0.100449 val accuracy: 0.078000
lr 1.000000e-03 reg 1.000000e-06 hidden_size 8.000000e+01 num_training_epochs
5.000000e+03 train accuracy: 0.100429 val accuracy: 0.079000
lr 1.000000e-02 reg 2.000000e-07 hidden_size 1.000000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.234898 val accuracy: 0.245000
lr 1.000000e-02 reg 2.000000e-07 hidden_size 2.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.328857 val accuracy: 0.322000
lr 1.000000e-02 reg 1.000000e-06 hidden_size 5.000000e+01 num_training_epochs
5.000000e+03 train accuracy: 0.292388 val accuracy: 0.293000
lr 1.000000e-02 reg 1.000000e-05 hidden_size 1.500000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.244898 val accuracy: 0.257000
lr 1.000000e-01 reg 1.000000e-07 hidden_size 1.000000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.579755 val accuracy: 0.555000
lr 1.000000e-01 reg 1.000000e-07 hidden_size 2.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.622020 val accuracy: 0.569000
lr 1.000000e-01 reg 2.000000e-07 hidden_size 1.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.608449 val accuracy: 0.581000
lr 1.000000e-01 reg 1.000000e-06 hidden_size 1.000000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.576714 val accuracy: 0.548000
lr 1.000000e-01 reg 1.000000e-05 hidden_size 5.000000e+01 num_training_epochs
5.000000e+03 train accuracy: 0.587898 val accuracy: 0.546000
lr 2.000000e-01 reg 1.000000e-07 hidden_size 1.500000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.648939 val accuracy: 0.569000
lr 2.000000e-01 reg 2.000000e-07 hidden_size 5.000000e+01 num_training_epochs
3.000000e+03 train accuracy: 0.606673 val accuracy: 0.567000
lr 2.000000e-01 reg 1.000000e-06 hidden_size 5.000000e+01 num_training_epochs
5.000000e+03 train accuracy: 0.622122 val accuracy: 0.565000
lr 2.000000e-01 reg 1.000000e-05 hidden_size 5.000000e+01 num_training_epochs
3.000000e+03 train accuracy: 0.601673 val accuracy: 0.564000
lr 2.000000e-01 reg 1.000000e-05 hidden_size 2.000000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.652776 val accuracy: 0.591000
lr 3.000000e-01 reg 1.000000e-07 hidden_size 3.000000e+01 num_training_epochs
5.000000e+03 train accuracy: 0.592796 val accuracy: 0.547000
lr 3.000000e-01 reg 2.000000e-07 hidden_size 1.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.689388 val accuracy: 0.572000
lr 3.000000e-01 reg 2.000000e-07 hidden_size 1.500000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.682653 val accuracy: 0.594000
lr 3.000000e-01 reg 1.000000e-06 hidden_size 5.000000e+01 num_training_epochs
5.000000e+03 train accuracy: 0.628939 val accuracy: 0.553000
lr 3.000000e-01 reg 1.000000e-06 hidden_size 1.500000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.722551 val accuracy: 0.586000
lr 3.000000e-01 reg 1.000000e-05 hidden_size 1.000000e+02 num_training_epochs
5.000000e+03 train accuracy: 0.684347 val accuracy: 0.582000
lr 3.000000e-01 reg 1.000000e-05 hidden_size 1.500000e+02 num_training_epochs
3.000000e+03 train accuracy: 0.679531 val accuracy: 0.582000
best validation accuracy achieved during cross-validation: 0.598000
```

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

```
0.579
```

---

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]:
```python
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()))
```