knn

October 8, 2022

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

     # Enter the foldername in your Drive where you have saved the unzipped
     # assignment folder, e.g. 'cs231n/assignments/assignment1/'
     FOLDERNAME = 'ENPM809K/Assignments/assignment1/'
     assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/Assignments/assignment1
```

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

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

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

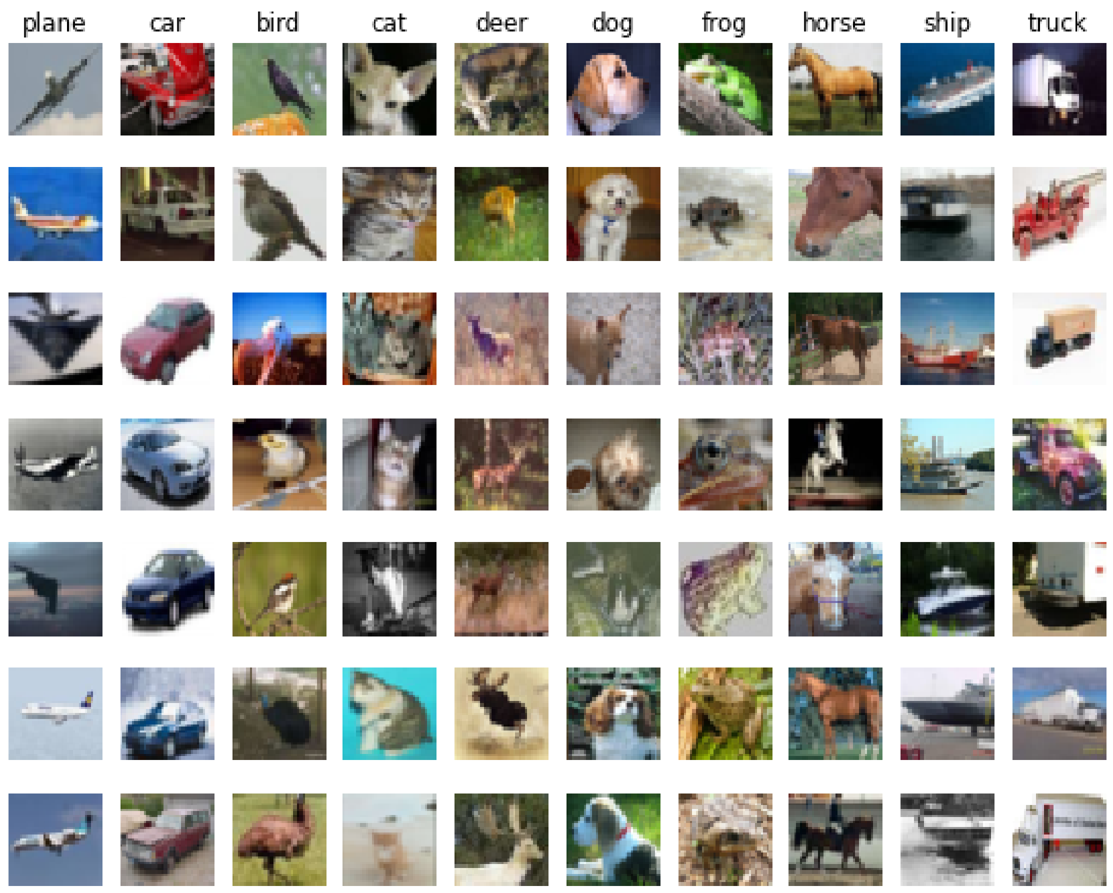
- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

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

[ ]:
```
# 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
```

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

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

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

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

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

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

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



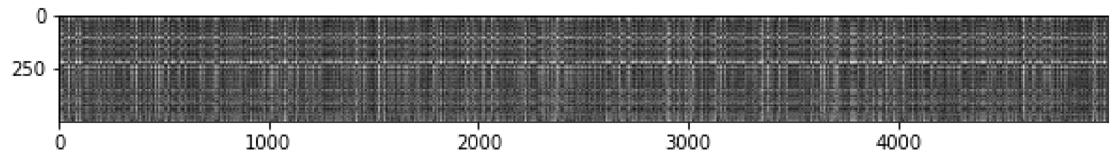
```
```````````````````````````````````````````````
```

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

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

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

```
print('``````````````````````````````````````````````')
```

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

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

```
||||||||||||||||||||||||||||||||||||||||||||||
~~~
```

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

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

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

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

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

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

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

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

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

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

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


0
250
0
1000
2000
3000
4000

```
~~~
```

Inline Question 1

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

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

YourAnswer : - The data on which we are working are pixel intensity values, distinctly bright rows indicate a significant delta in pixel intensities between a single test image and rest of all training images .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 i.e., if all the test images are very different from a single training image, the column will be bright.

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

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
```
~~~

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

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

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

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

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

## 2 New Section

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

**Inline Question 2**

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

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

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean $\mu$ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.) 2. Subtracting the per pixel mean $\mu_{ij}$ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean $\mu$ and dividing
~~~

by the standard deviation 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.

$\textit{Your Answer :}$ 1, 2 and 3

$\textit{Your Explanation :}$ 1. As all the pixel values are constrained to [0,255]. Subtracting the mean, will just offset the pixel values and won't have drastic change in the performance of classifier.

It can be showed mathematically as

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

Here the μ, gets cancelled out. The equation shows the L1 distance between a single test image and corresponding train images. 2. Similarly, subtracting mean per pixel from image, won't change the performace of classifier as itsjust simple offseting of data. 3. Subtracting the data by a constant mean and dividing by single value of standard deviation is equivalent to converting into normal distribution.The L1 distance is effectively just being scaled by a fixed factor. This would cause the nearest neighbors of a point to remain the same . 4. Dividing by pixel-wise standard deviation, the different dimensions are scaled differently. Hence, affecting the performance of the classifier. 5. Rotation doesn't affect L2 distances. But it significantly affect L1 distance.

```
[ ]: # PLEASE DO NOT MODIFY THE MARKERS
     print('||||||||||||||||||||||||||||||||||||||||||||||')
     # Now lets speed up distance matrix computation by using partial vectorization
     # with one loop. Implement the function compute_distances_one_loop and run the
     # code below:
     dists_one = classifier.compute_distances_one_loop(X_test)

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

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

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

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

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

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

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

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

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

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

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

```
|||||||||||||||||||||||||||||||||||||||||||||||
Two loop version took 40.772807 seconds
One loop version took 32.491033 seconds
No loop version took 0.520259 seconds
```````````````````````````````````````````````
```

2.0.1 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.

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

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

# The training set is divided into 'num_folds' parts equally.
# 'vsplit' was used to vertically split the training data (2D array).
# 'split' for 'y_train' as it is 1D array.

X_train_folds = np.vsplit(X_train , num_folds)
y_train_folds = np.split(y_train , num_folds)

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

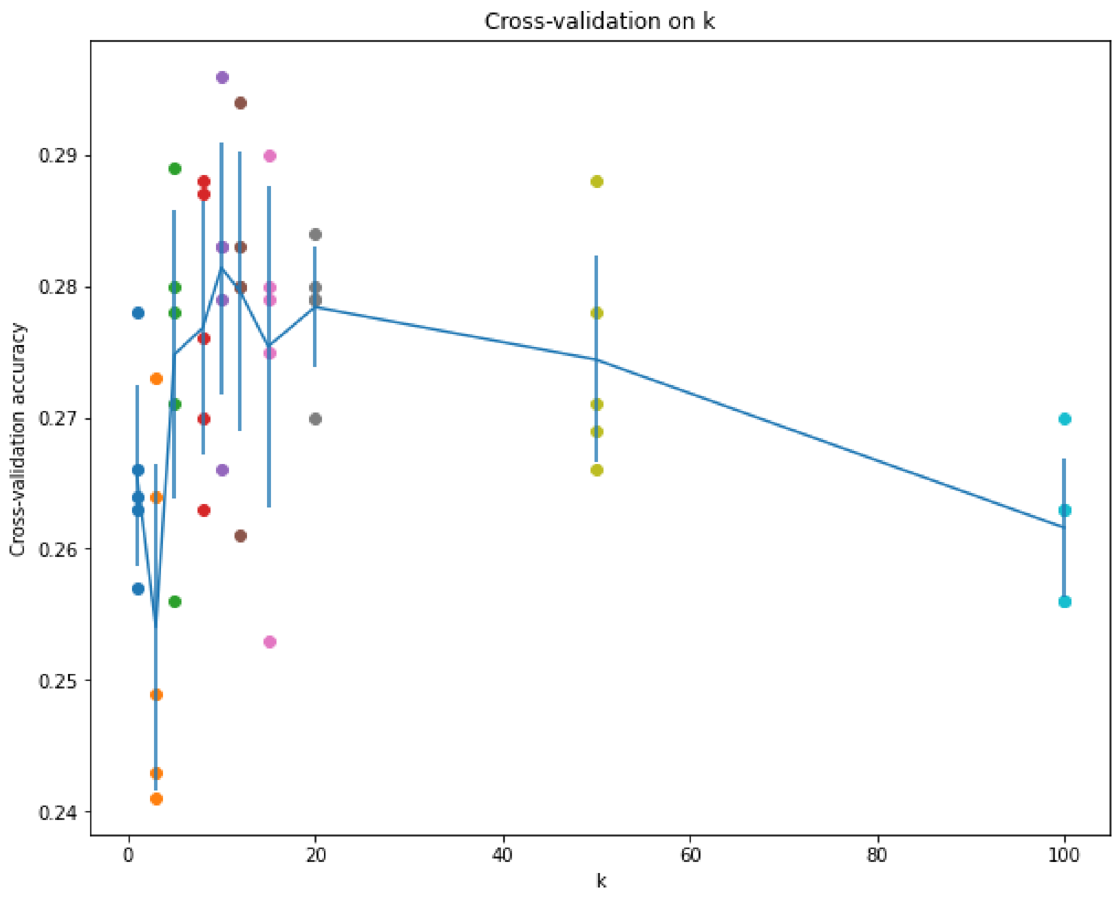
# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}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

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

# The first 'for' loop for choosing the value of 'k'.
# The second 'for' loop for choosing different validation set from training data
# for the chosen 'k' and computes accuracy.
for k in k_choices:

  k_to_accuracies[k] = []

  for i in range(num_folds):
    # A copy is made of both lists.
    X_copy = X_train_folds.copy()
    y_copy = y_train_folds.copy()
    # The validation set is chosen from the copy oftraining set
    X_validation_fold = X_copy[i]
    y_validation_fold = y_copy[i]

    # The validation set is deleted from the copy of training set.
    X_copy.pop(i)

    # The remaining elements are stacked together to form the training set and
    # converted into numpy array
    X_trainn = np.vstack((X_copy[:]))

    # The same pop and stack method is used for labels as well.
    y_copy.pop(i)
    y_trainn = np.concatenate( (y_copy[:]) , axis =None)

    # A new classifier is defined and trained using 'X_trainn' and 'y_trainn'.
    classifier_new = KNearestNeighbor()
    classifier_new.train( X_trainn, y_trainn )
    # Distances calculated between validation set and training set.
    dists_new = classifier_new.compute_distances_no_loops( X_validation_fold )
    # Prediction takes place for chosen value of k neighbours.
    y_test_pred_new = classifier_new.predict_labels( dists_new, k=k )
    # Accuracy is calculated for the predicted data and stored into the␣
↪dictionary.
    num_correct = np.sum(y_test_pred_new == y_validation_fold)
    accuracy = float(num_correct) / X_validation_fold.shape[0]

    k_to_accuracies[k].append(accuracy)


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

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

```
||||||||||||||||||||||||||||||||||||||||||||||
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.241000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.243000
k = 3, accuracy = 0.273000
k = 3, accuracy = 0.264000
k = 5, accuracy = 0.256000
k = 5, accuracy = 0.271000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.289000
k = 5, accuracy = 0.278000
k = 8, accuracy = 0.263000
k = 8, accuracy = 0.287000
k = 8, accuracy = 0.276000
k = 8, accuracy = 0.288000
k = 8, accuracy = 0.270000
k = 10, accuracy = 0.266000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.279000
k = 10, accuracy = 0.283000
k = 10, accuracy = 0.283000
k = 12, accuracy = 0.261000
k = 12, accuracy = 0.294000
k = 12, accuracy = 0.280000
k = 12, accuracy = 0.283000
```

```
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.253000
k = 15, accuracy = 0.290000
k = 15, accuracy = 0.279000
k = 15, accuracy = 0.280000
k = 15, accuracy = 0.275000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.280000
k = 20, accuracy = 0.284000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
``````````````````````````````````````````````
```

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

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

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



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

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

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

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

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

**Inline Question 3**

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

$\color{blue}{\textit{Your Answer :}}$ 2 and 4 are True

$\color{blue}{\textit{Your Explanation :}}$ 1. The decision boundary of knn classifier need not be linear always. 2. For 1-NN classifier the nearest neighbour would be itself, therby making the training error 0. 3. It varies for different values of k, as evident from the exercise. Cross validation is the best way to choose k with less test error.
4. k-NN essentially computes distance between a test image and entire training set. So if the training set is really huge, it would take more time to predict the label.

# svm

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/Assignments/assignment1
```

## 1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

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

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

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

### 1.1 CIFAR-10 Data Loading and Preprocessing

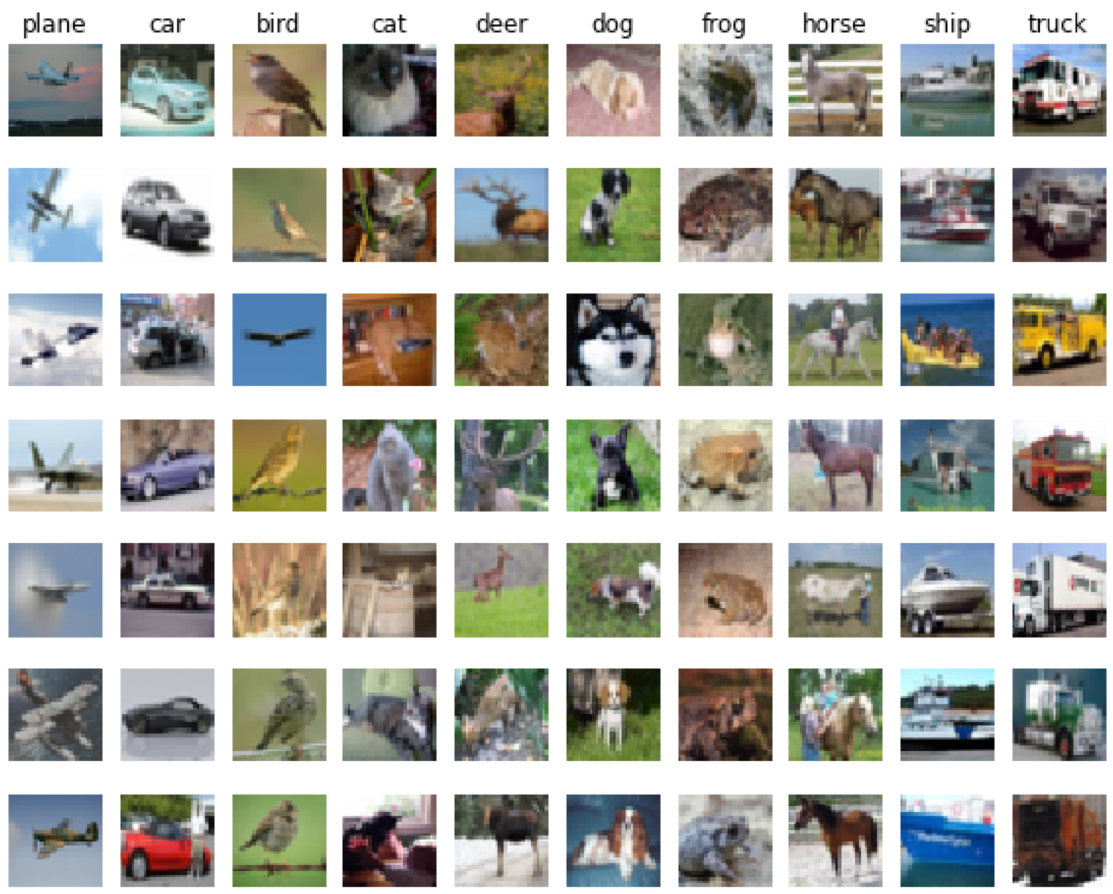
```
[ ]: # Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

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

[ ]:

```
# 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()
```



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

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

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

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

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

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

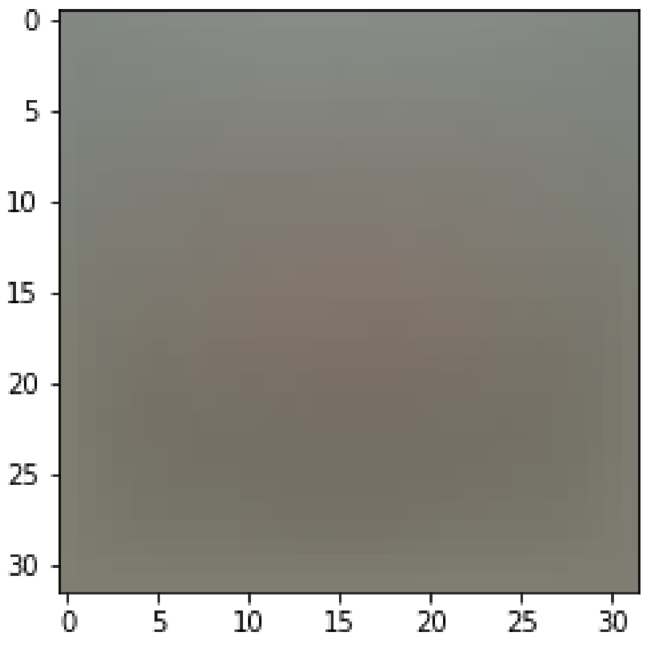
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

```
[ ]: # 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)
```

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

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

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

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

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



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

### 1.2 SVM Classifier

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

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

```
[ ]: # 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: 8.949048
```

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:

```
[ ]: # 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: -18.608638 analytic: -18.608638, relative error: 3.868274e-12
numerical: 28.497918 analytic: 28.497918, relative error: 6.305156e-12
numerical: -41.339494 analytic: -41.339494, relative error: 4.041755e-12
numerical: 14.691799 analytic: 14.691799, relative error: 8.835417e-12
numerical: 21.420011 analytic: 21.420011, relative error: 4.032295e-12
numerical: -4.404608 analytic: -4.404608, relative error: 6.268386e-11
numerical: 3.385214 analytic: 3.385214, relative error: 9.750901e-11
numerical: -24.131808 analytic: -24.131808, relative error: 1.553375e-11
numerical: -14.587609 analytic: -14.587609, relative error: 1.741060e-11
numerical: 12.613461 analytic: 12.613461, relative error: 2.846088e-11
numerical: 11.094667 analytic: 11.094667, relative error: 2.446010e-11
numerical: -5.528970 analytic: -5.528970, relative error: 3.008787e-11
numerical: 5.037235 analytic: 5.037235, relative error: 1.518885e-10
numerical: -20.174183 analytic: -20.174183, relative error: 4.838490e-12
numerical: 23.843650 analytic: 23.843650, relative error: 9.043133e-12
numerical: 13.027216 analytic: 13.027216, relative error: 1.442109e-11
numerical: -12.368388 analytic: -12.368388, relative error: 1.968171e-11
numerical: -30.096259 analytic: -30.096259, relative error: 6.188511e-12
numerical: 1.150110 analytic: 1.150110, relative error: 1.818430e-10
numerical: -1.272948 analytic: -1.272948, relative error: 3.879137e-10
```

**Inline Question 1**

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

$Your Answer:$ * The hinge loss for the SVM function is:

$$f(x) = max(0, x)$$

, where $x$ is the difference between the scores of incorrect classes and correct class plus a constant.

If $x > 0$, then the function attributes to a error and added to the loss, whereas $x < 0$ is discarded by assigning 0. The gradient of hinge loss is undefined at x =0(kink). In general for a function $f(x) = max(x, y)$ ,the gradient is undefined at $x = y$, these non-differentiable parts of the function are known as kinks and they are the cause of fails in gradient check. * For a simple example, take $x = -e^{-7}$ then $max(0, x) = 0$, the analytical gradient is 0. But, the numerical gradient might be different if we consider h greater than x, like $h = 1e^{-3}$. * As the margin increases, it increases the chances of the scores being higher. Thus, $max(0, x)$ would output positive numbers more frequently and reduces the risk of gradient mismatch at x=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: 8.949048e+00 computed in 0.140660s
Vectorized loss: 8.949048e+00 computed in 0.020235s
difference: 0.000000
```

[ ]:
```
# 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 0.120374s
Vectorized loss and gradient: computed in 0.031467s
difference: 0.000000
```

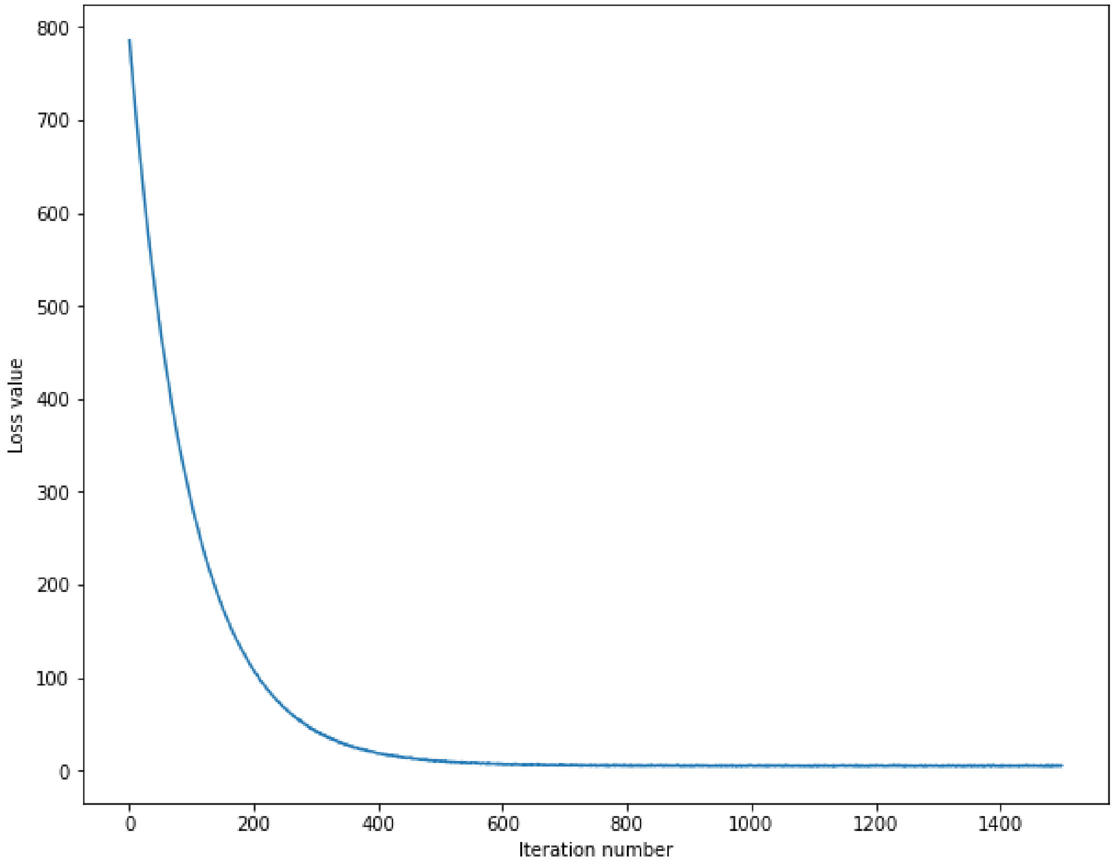
### 1.2.1 Stochastic Gradient Descent

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`.

[ ]:
```
# 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 785.739459
iteration 100 / 1500: loss 287.036696
iteration 200 / 1500: loss 107.506118
iteration 300 / 1500: loss 42.469339
iteration 400 / 1500: loss 18.809344
iteration 500 / 1500: loss 10.690896
iteration 600 / 1500: loss 6.601916
iteration 700 / 1500: loss 5.735955
iteration 800 / 1500: loss 5.290206
iteration 900 / 1500: loss 5.374317
iteration 1000 / 1500: loss 6.008256
iteration 1100 / 1500: loss 5.753323
iteration 1200 / 1500: loss 5.099108
iteration 1300 / 1500: loss 4.991404
iteration 1400 / 1500: loss 5.118442
That took 10.181741s
```

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



[ ]:
```
# 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: 0.364796
validation accuracy: 0.373000
```

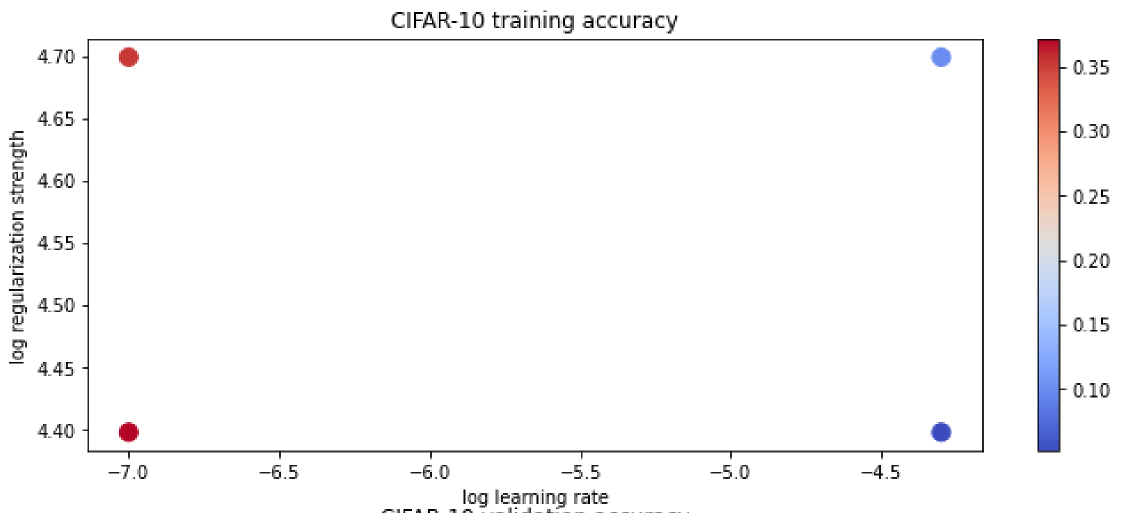
[ ]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation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

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

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

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
# learning_rates = [1e-7, 5e-5 , 1e-6 ,1.5e-7]
# regularization_strengths = [2.5e4, 5e4 , 7e3 , 3.5e4]

learning_rates = [1e-7, 5e-5 ]
regularization_strengths = [2.5e4, 5e4 ]
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Storing all possible combinations of learning rates and regularization␣
 ↪constants
possible_combs = []
for i in learning_rates:
  for j in regularization_strengths:
    possible_combs.append((i,j))


for l_rate, r_strength in possible_combs:
    # Initialize SVM object
    svm_new = LinearSVM()
    # training and predicting for the current choice of l_rate and r_strength on
    # the training set
    train_loss = svm_new.train(X_train, y_train, learning_rate= l_rate , reg=␣
 ↪r_strength,
                        num_iters=1500, verbose=False)
    y_train_pred = svm_new.predict(X_train)

    #Getting training data acc and applying model to validation data and storing
    # its accuracy
    train_accuracy = np.mean(y_train_pred == y_train)

    y_val_pred = svm_new.predict(X_val)

    val_accuracy = np.mean(y_val_pred == y_val)
    #Storing the results and selecting the object with best validation accuracy
    results[(l_rate , r_strength)] = (train_accuracy, val_accuracy)
    if best_val < val_accuracy:
        best_val = val_accuracy
        best_svm = svm_new

# *****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)
```

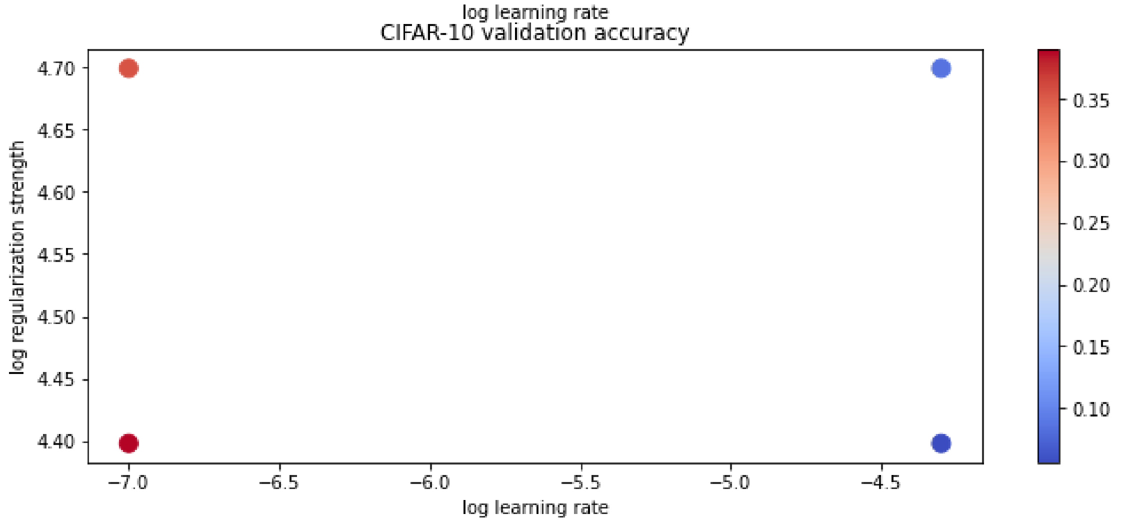
```
/content/drive/My
Drive/ENPM809K/Assignments/assignment1/cs231n/classifiers/linear_svm.py:113:
RuntimeWarning: overflow encountered in double_scalars
  loss += reg * np.sum(W * W)
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:86:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/drive/My
Drive/ENPM809K/Assignments/assignment1/cs231n/classifiers/linear_svm.py:113:
RuntimeWarning: overflow encountered in multiply
  loss += reg * np.sum(W * W)
/content/drive/My
Drive/ENPM809K/Assignments/assignment1/cs231n/classifiers/linear_svm.py:141:
RuntimeWarning: overflow encountered in multiply
```

```
  dW += 2 * reg * W
/content/drive/My Drive/ENPM809K/Assignments/assignment1/cs231n/classifiers/line
ar_classifier.py:91: RuntimeWarning: invalid value encountered in subtract
  self.W = self.W - learning_rate*grad
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.372000 val accuracy: 0.390000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.354429 val accuracy: 0.356000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.051755 val accuracy: 0.056000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.390000
```

[ ]:
```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()
```



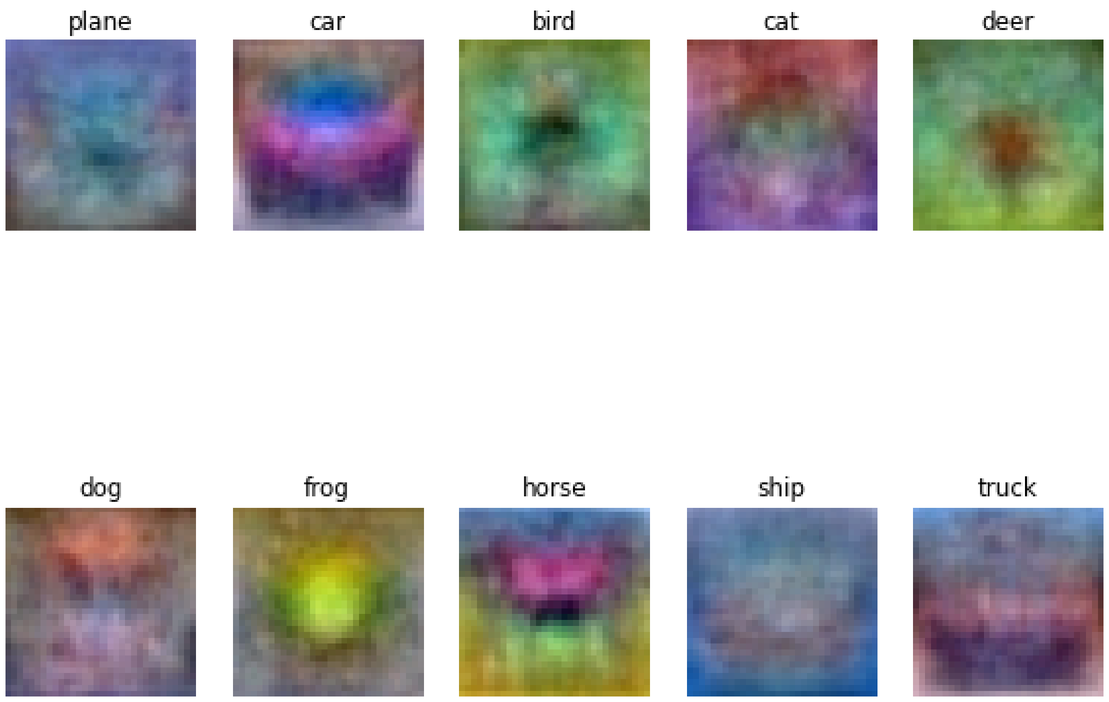


[ ]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.371000
```

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

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

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



**Inline question 2**

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

*Your Answer* : Each class weight seems to hold image information that is attributed to find that particular class. Like in car class, we can make out a blurred out red car. And in class 'ship', the background is more blue in color to signify water presence. Each class has this distinct feature , so we only need to compare a single set of weights with the test dataset rather than compare with the entire training set like in KNN.

The appearance of the classes is because of the combination of all available training set provided to make weights. The weight class finds out a general mask, which when computed with similar test images will make predictions accordingly.

# softmax

October 8, 2022

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

     # Enter the foldername in your Drive where you have saved the unzipped
     # assignment folder, e.g. 'cs231n/assignments/assignment1/'
     FOLDERNAME = 'ENPM809K/Assignments/assignment1/'
     assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/Assignments/assignment1
```

## 1 Softmax exercise

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

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

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

- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

[ ]:
```
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
```

[ ]:
```
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`.

```
[ ]: # 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: 2.350289
sanity check: 2.302585
```

**Inline Question 1**

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

$Your Answer:$ At the beginning all the weights are really small and can be assumed as zero. This makes all the scores to zero. And since each class makes upto 10% of the dataset, the probability of choosing an image and predicting it would be of a particular class would be 0.1. Hence, loss close to -log(0.1)

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

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

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

```
numerical: 0.023740 analytic: 0.023740, relative error: 7.426555e-07
numerical: -0.729025 analytic: -0.729025, relative error: 2.045177e-08
numerical: -2.288217 analytic: -2.288217, relative error: 2.696022e-08
numerical: 1.788377 analytic: 1.788377, relative error: 9.209871e-09
numerical: 0.401405 analytic: 0.401405, relative error: 2.316821e-07
numerical: -2.034935 analytic: -2.034935, relative error: 1.654710e-08
numerical: 2.069318 analytic: 2.069318, relative error: 1.023562e-08
numerical: -0.143941 analytic: -0.143941, relative error: 2.880922e-07
numerical: -1.793882 analytic: -1.793882, relative error: 2.104597e-08
numerical: -2.138923 analytic: -2.138923, relative error: 1.665470e-08
numerical: -0.819290 analytic: -0.819290, relative error: 1.204983e-08
numerical: 0.119463 analytic: 0.119463, relative error: 2.286152e-07
numerical: 0.370427 analytic: 0.370427, relative error: 1.403869e-07
numerical: 1.539087 analytic: 1.539087, relative error: 5.639427e-08
numerical: -1.686440 analytic: -1.686440, relative error: 7.656221e-09
numerical: -0.809756 analytic: -0.809756, relative error: 2.704124e-08
numerical: -0.538436 analytic: -0.538436, relative error: 1.315705e-09
numerical: -2.201244 analytic: -2.201244, relative error: 1.934768e-08
numerical: -0.264227 analytic: -0.264227, relative error: 1.373903e-07
numerical: 1.181716 analytic: 1.181716, relative error: 6.699426e-08
```

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

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

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

```
naive loss: 2.350289e+00 computed in 0.100746s
vectorized loss: 2.350289e+00 computed in 0.015441s
```

```
Loss difference: 0.000000
Gradient difference: 0.000000
```

[ ]:
```
# 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-7, 5e-7 , 1e-6 ,1.5e-7]
regularization_strengths = [2.5e4, 5e4 , 7e3 , 3.5e4]
# learning_rates = [1e-7, 5e-7 ]
# regularization_strengths = [2.5e4, 5e4]
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Storing all possible combinations of learning rates and regularization␣
↪constants
possible_combs = []
for i in learning_rates:
  for j in regularization_strengths:
    possible_combs.append((i,j))


for l_rate, r_strength in possible_combs:
    # Initialize Softmax object
    soft_new = Softmax()
    # training and predicting for the current choice of l_rate and r_strength on
    # the training set
    train_loss = soft_new.train(X_train, y_train, learning_rate= l_rate , reg=␣
↪r_strength,
                        num_iters=1500, verbose=False)
    y_train_pred = soft_new.predict(X_train)

    #Getting training data acc and applying model to validation data and storing
    # its accuracy
    train_accuracy = np.mean(y_train_pred == y_train)

    y_val_pred = soft_new.predict(X_val)

    val_accuracy = np.mean(y_val_pred == y_val)

    #Storing the results and selecting the object with best validation accuracy
    results[(l_rate , r_strength)] = (train_accuracy, val_accuracy)
    if best_val < val_accuracy:
        best_val = val_accuracy
        best_softmax = soft_new


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

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

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

```
lr 1.000000e-07 reg 7.000000e+03 train accuracy: 0.348102 val accuracy: 0.361000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.331837 val accuracy: 0.344000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.319061 val accuracy: 0.325000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.313939 val accuracy: 0.328000
lr 1.500000e-07 reg 7.000000e+03 train accuracy: 0.362837 val accuracy: 0.376000
lr 1.500000e-07 reg 2.500000e+04 train accuracy: 0.324224 val accuracy: 0.343000
lr 1.500000e-07 reg 3.500000e+04 train accuracy: 0.324469 val accuracy: 0.339000
lr 1.500000e-07 reg 5.000000e+04 train accuracy: 0.299388 val accuracy: 0.319000
lr 5.000000e-07 reg 7.000000e+03 train accuracy: 0.361959 val accuracy: 0.376000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.323531 val accuracy: 0.340000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.325020 val accuracy: 0.334000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.304408 val accuracy: 0.315000
lr 1.000000e-06 reg 7.000000e+03 train accuracy: 0.359857 val accuracy: 0.385000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.327816 val accuracy: 0.342000
lr 1.000000e-06 reg 3.500000e+04 train accuracy: 0.312469 val accuracy: 0.329000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.283959 val accuracy: 0.304000
best validation accuracy achieved during cross-validation: 0.385000
```

[ ]:
```
# 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: 0.351000
```

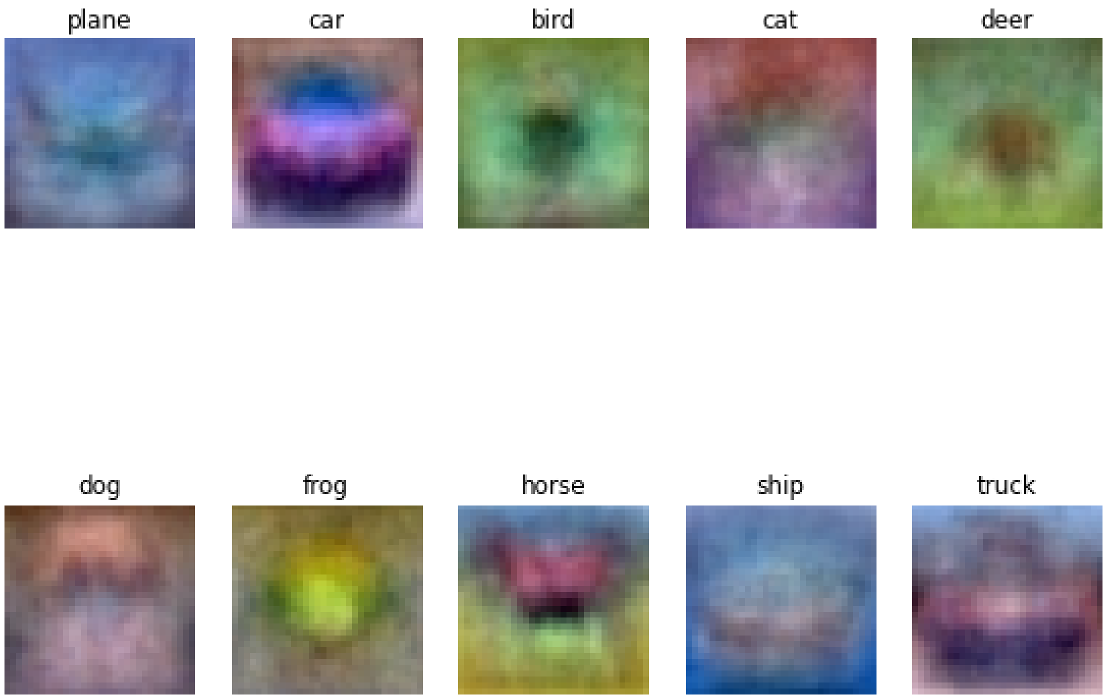
**Inline Question 2** - *True or False*

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

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

$\color{blue}{\textit{Your Explanation :}}$
* SVM loss function only cares whether the correct class score is higher than rest of the classes by delta. For instance, scores = [10,8,5] and *delta*=1. Here 10 is the correct class score. and if another datapoint is added such that scores = [10,8,5,7]. The loss funcion does not change as the data is already outputting correct class. * Whereas in Softmax, addition of a datapoint effects the loss function. The model tries to minimise the influence of other class scores, even when the probability of correct class is higher than rest. the softmax problem tries to improve the correct class probability to 1 and others to 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])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

[ ]:

# two_layer_net

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/Assignments/assignment1
```

## 1 Fully-Connected Neural Nets

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

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

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

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

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

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

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

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

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

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

## 2 Affine layer: forward

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

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

```
[ ]: # Test the affine_forward function

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

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

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

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

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

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

## 3 Affine layer: backward

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

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

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x,␣
 ↪dout)
print(dx_num.shape)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w,␣
 ↪dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b,␣
 ↪dout)

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

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

```
(10, 2, 3)
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

## 4 ReLU activation: forward

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

[ ]:
```
# Test the relu_forward function

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

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

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

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

## 5 ReLU activation: backward

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

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

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

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

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

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

### 5.1 Inline Question 1:

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

### 5.2 Answer:

The functions with zero gradient flow problem (1) Sigmoid and (2) ReLu. ### Reason: * For Sigmoid, when the neuron's activation becomes close to 0 or 1, the gradient is 0. Mostly caused by unnormalised weights. * In case of ReLu, when the input values cause the activation tobe 0. The gradient with respect inputs would be 0.

#### 5.2.1 Examples:

- Sigmoid - Take a very large input (+ve / -ve), [eg: $10^7$], then the formula for sigmoid evaluates to 0 or 1. $\sigma(x) = \frac{1}{1+e^{-x}}$. And the derivative with respect to the feature evaluates to $(1 - \sigma(x))\sigma(x)$, which leads to zero either way.
- ReLu - The function is $f(x) = max(0, x)$ , and when any -ve negative value(eg: -0.7) is inputted it evaluates to zero. Therefore, the gradient will be zero as well.

## 6 "Sandwich" layers

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

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

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

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

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

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

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
```

```
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

## 7 Loss layers: Softmax and SVM

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

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

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

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

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

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

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

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-09
```

## 8 Two-layer network

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

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

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

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

     print('Testing test-time forward pass ... ')
     model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
     model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
     model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
     model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
     X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
     scores = model.loss(X)
     correct_scores = np.asarray(
       [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
      ↪33206765,  16.09215096],
        [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
      ↪49994135,  16.18839143],
        [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
      ↪66781506,  16.2846319 ]])
     scores_diff = np.abs(scores - correct_scores).sum()
     assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

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

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.20e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

## 9 Solver

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

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

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                            #
##############################################################################
```

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Setting up an solver instance and training the model
solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
solver.train()

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

```
(Iteration 1 / 4900) loss: 2.300089
(Epoch 0 / 10) train acc: 0.171000; val_acc: 0.170000
(Iteration 101 / 4900) loss: 1.782419
(Iteration 201 / 4900) loss: 1.803466
(Iteration 301 / 4900) loss: 1.712676
(Iteration 401 / 4900) loss: 1.693946
(Epoch 1 / 10) train acc: 0.399000; val_acc: 0.428000
(Iteration 501 / 4900) loss: 1.711237
(Iteration 601 / 4900) loss: 1.443683
(Iteration 701 / 4900) loss: 1.574904
(Iteration 801 / 4900) loss: 1.526751
(Iteration 901 / 4900) loss: 1.352554
(Epoch 2 / 10) train acc: 0.463000; val_acc: 0.443000
(Iteration 1001 / 4900) loss: 1.340071
(Iteration 1101 / 4900) loss: 1.386843
(Iteration 1201 / 4900) loss: 1.489919
(Iteration 1301 / 4900) loss: 1.353077
(Iteration 1401 / 4900) loss: 1.467951
(Epoch 3 / 10) train acc: 0.477000; val_acc: 0.430000
(Iteration 1501 / 4900) loss: 1.337133
(Iteration 1601 / 4900) loss: 1.426819
(Iteration 1701 / 4900) loss: 1.348675
(Iteration 1801 / 4900) loss: 1.412626
(Iteration 1901 / 4900) loss: 1.354764
(Epoch 4 / 10) train acc: 0.497000; val_acc: 0.467000
(Iteration 2001 / 4900) loss: 1.422221
(Iteration 2101 / 4900) loss: 1.360665
(Iteration 2201 / 4900) loss: 1.546539
(Iteration 2301 / 4900) loss: 1.196704
(Iteration 2401 / 4900) loss: 1.401958
```

```
(Epoch 5 / 10) train acc: 0.508000; val_acc: 0.483000
(Iteration 2501 / 4900) loss: 1.243567
(Iteration 2601 / 4900) loss: 1.513808
(Iteration 2701 / 4900) loss: 1.266416
(Iteration 2801 / 4900) loss: 1.485956
(Iteration 2901 / 4900) loss: 1.049706
(Epoch 6 / 10) train acc: 0.533000; val_acc: 0.492000
(Iteration 3001 / 4900) loss: 1.264002
(Iteration 3101 / 4900) loss: 1.184786
(Iteration 3201 / 4900) loss: 1.231162
(Iteration 3301 / 4900) loss: 1.353725
(Iteration 3401 / 4900) loss: 1.217830
(Epoch 7 / 10) train acc: 0.545000; val_acc: 0.489000
(Iteration 3501 / 4900) loss: 1.446003
(Iteration 3601 / 4900) loss: 1.152495
(Iteration 3701 / 4900) loss: 1.421970
(Iteration 3801 / 4900) loss: 1.222752
(Iteration 3901 / 4900) loss: 1.213468
(Epoch 8 / 10) train acc: 0.546000; val_acc: 0.474000
(Iteration 4001 / 4900) loss: 1.187435
(Iteration 4101 / 4900) loss: 1.284799
(Iteration 4201 / 4900) loss: 1.135251
(Iteration 4301 / 4900) loss: 1.212217
(Iteration 4401 / 4900) loss: 1.213544
(Epoch 9 / 10) train acc: 0.586000; val_acc: 0.486000
(Iteration 4501 / 4900) loss: 1.306174
(Iteration 4601 / 4900) loss: 1.213528
(Iteration 4701 / 4900) loss: 1.220260
(Iteration 4801 / 4900) loss: 1.233231
(Epoch 10 / 10) train acc: 0.545000; val_acc: 0.483000
```

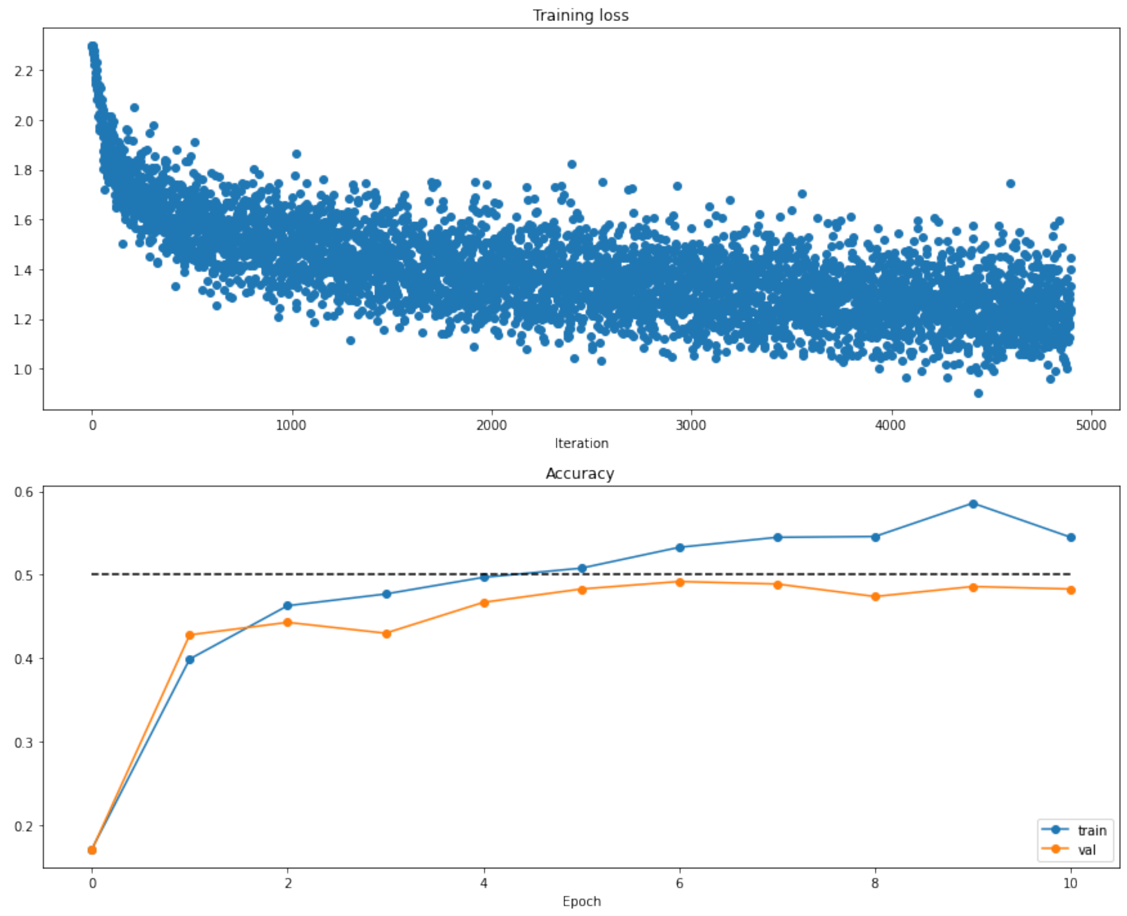
## 10 Debug the training

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

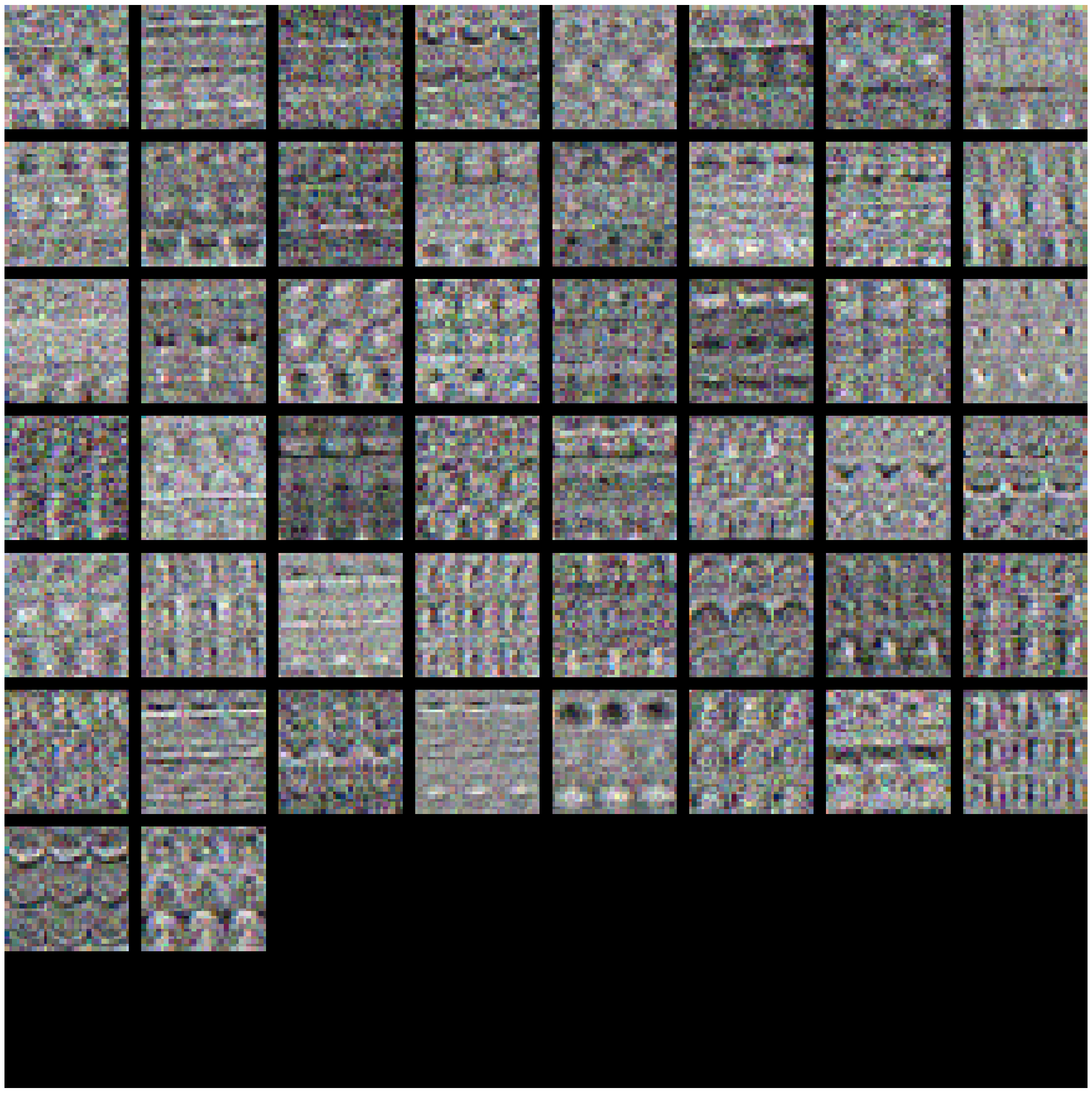
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

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

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

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



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

# Visualize the weights of the network

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

show_net_weights(model)
```



## 11 Tune your hyperparameters

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

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

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

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

[ ]:
```
best_model = None


#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
 ↪#
# model in best_model.                                                         ␣
 ↪#
#                                                                              ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative   ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
 ↪#
#                                                                              ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters         ␣
 ↪#
# automatically like we did on thexs previous exercises.                       ␣
 ↪  #
```

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

# Setting intial parameters and dimension for the model
best_val_accuracy = 0.0
results = {}

input_size = 32 * 32 * 3
num_classes = 10
hidden_size = 77
# Different learning and regularization strengths initialised
# learning_rates = [1e-3 , 2e-3, 3e-2  ]
# regularization_strengths = [0.3 , 0.7 , 4 , 1.7]
learning_rates = [1e-3 , 3e-4   ,  1e-4, 2e-3]
regularization_strengths = [1e-4 , 0.075    , 1.5e-5 , 0.7]

# Storing all possible combinations of learning rates and regularization␣
 ↪constants
possible_combs = []
for i in learning_rates:
  for j in regularization_strengths:
    possible_combs.append((i,j))


for l_rate, r_strength in possible_combs:
    solver = None
    # Two layered network formed with the given parametrs
    model = TwoLayerNet(input_size, hidden_size, num_classes, reg = r_strength)
    # Solver object initialised and the trained
    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': l_rate,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                     verbose = False)
    solver.train()

    # Calcualting the validation accuracy for each pair of learning and␣
 ↪regularization
    # rate, and storing the model with highest accuracy
    results[(l_rate , r_strength)] = solver.best_val_acc
    if solver.best_val_acc > best_val_accuracy:
        best_val_accuracy = solver.best_val_acc
        best_model = model
```

```
for lr, reg in sorted(results):
    val_accuracy = results[(lr, reg)]
    print('lr %e reg %e val accuracy: %f' % (
                lr, reg,  val_accuracy))

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

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

```
lr 1.000000e-04 reg 1.500000e-05 val accuracy: 0.464000
lr 1.000000e-04 reg 1.000000e-04 val accuracy: 0.467000
lr 1.000000e-04 reg 7.500000e-02 val accuracy: 0.466000
lr 1.000000e-04 reg 7.000000e-01 val accuracy: 0.470000
lr 3.000000e-04 reg 1.500000e-05 val accuracy: 0.506000
lr 3.000000e-04 reg 1.000000e-04 val accuracy: 0.508000
lr 3.000000e-04 reg 7.500000e-02 val accuracy: 0.524000
lr 3.000000e-04 reg 7.000000e-01 val accuracy: 0.512000
lr 1.000000e-03 reg 1.500000e-05 val accuracy: 0.516000
lr 1.000000e-03 reg 1.000000e-04 val accuracy: 0.521000
lr 1.000000e-03 reg 7.500000e-02 val accuracy: 0.518000
lr 1.000000e-03 reg 7.000000e-01 val accuracy: 0.503000
lr 2.000000e-03 reg 1.500000e-05 val accuracy: 0.502000
lr 2.000000e-03 reg 1.000000e-04 val accuracy: 0.502000
lr 2.000000e-03 reg 7.500000e-02 val accuracy: 0.506000
lr 2.000000e-03 reg 7.000000e-01 val accuracy: 0.482000
best validation accuracy achieved during cross-validation: 0.524000
```

## 12 Test your model!

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

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

```
Validation set accuracy:  0.524
```

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

```
Test set accuracy:  0.503
```

## 12.1 Inline Question 2:

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

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

*Your Answer* : 1 and 3

*Your Explanation* :

1. [TRUE] : Provided the dataset has more variation, as this prevents overfitting. Adding more smaples to each class, expands the diversity of class variation and thereby the model generalizes better.
2. [FALSE] : The higher training accuracy can be interpreted as the model being able to detect complex patterns within the given training set. Adding more hidden dimensions lead to more training accuracy and might affect adversely on the test accuracy.
3. [TRUE] : It reduces the risk of overfitting on the training data, therby distributes the weights across all features as much as possible.

```
[ ]:
```

# features

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/Assignments/assignment1
```

## 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.

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


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

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

### 1.1 Load data

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

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

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

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

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

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

```
    X_test = X_test[mask]
    y_test = y_test[mask]

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

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

### 1.2 Extract Features

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

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

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

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

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

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

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

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

```python
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.

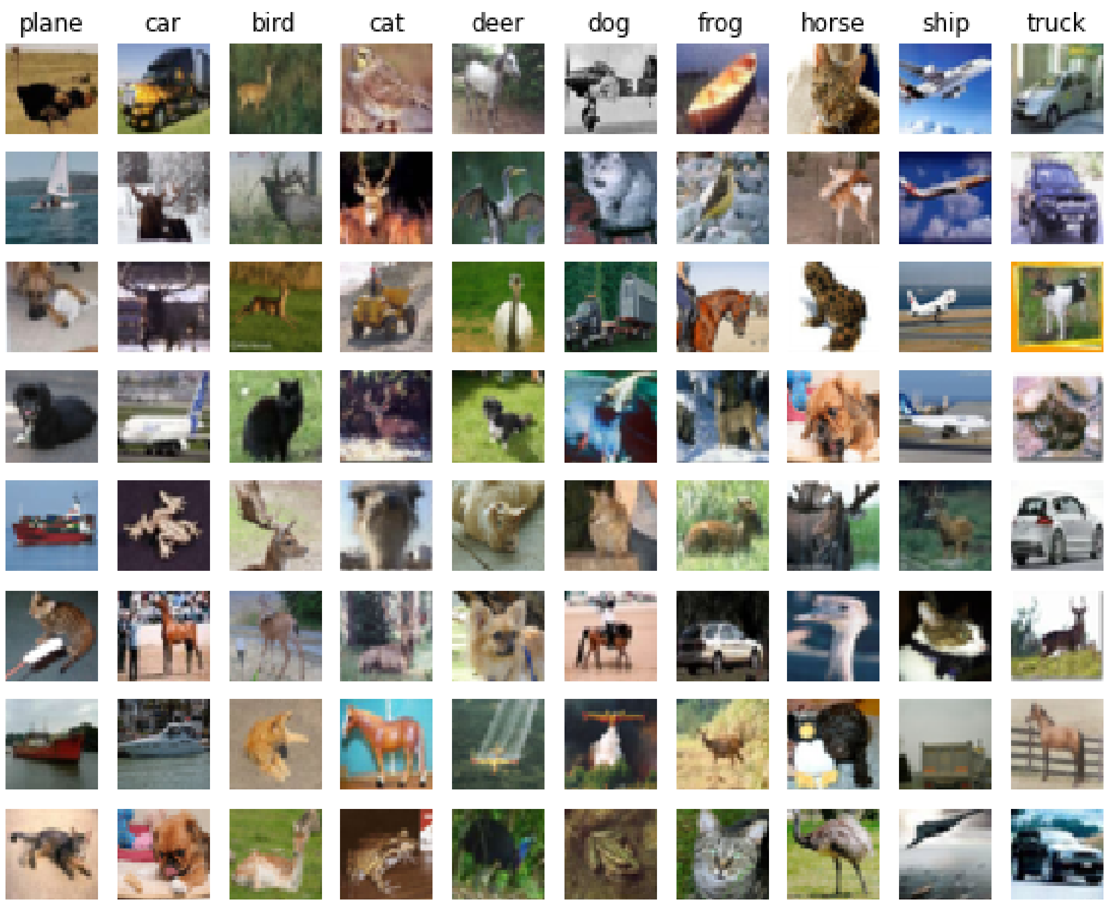
[5]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-10  , 7e-9 , 9e-9 , 3e-8]
regularization_strengths = [5e4, 5e5, 5e6 ,6e5  , 5e2]

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)*****

# Storing all possible combinations of learning rates and regularization␣
 ↪constants
possible_combs = []
for i in learning_rates:
  for j in regularization_strengths:
    possible_combs.append((i,j))


for l_rate, r_strength in possible_combs:
    # Initialize SVM object
    svm_new = LinearSVM()
    # training and predicting for the current choice of l_rate and r_strength on
    # the training set
    train_loss = svm_new.train(X_train_feats, y_train, learning_rate= l_rate ,␣
↪reg= r_strength,
                      num_iters=1500, verbose=False)
    y_train_pred = svm_new.predict(X_train_feats)

    #Getting training data acc and applying model to validation data and storing
    # its accuracy
    train_accuracy = np.mean(y_train_pred == y_train)

    y_val_pred = svm_new.predict(X_val_feats)

    val_accuracy = np.mean(y_val_pred == y_val)
    #Storing the results and selecting the object with best validation accuracy
    results[(l_rate , r_strength)] = (train_accuracy, val_accuracy)
    if best_val < val_accuracy:
        best_val = val_accuracy
        best_svm = svm_new

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

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

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

```
lr 1.000000e-10 reg 5.000000e+02 train accuracy: 0.098776 val accuracy: 0.082000
lr 1.000000e-10 reg 5.000000e+04 train accuracy: 0.107878 val accuracy: 0.087000
lr 1.000000e-10 reg 5.000000e+05 train accuracy: 0.093388 val accuracy: 0.080000
lr 1.000000e-10 reg 6.000000e+05 train accuracy: 0.089000 val accuracy: 0.083000
lr 1.000000e-10 reg 5.000000e+06 train accuracy: 0.093163 val accuracy: 0.113000
lr 1.000000e-09 reg 5.000000e+02 train accuracy: 0.101102 val accuracy: 0.093000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.095755 val accuracy: 0.114000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.092469 val accuracy: 0.093000
lr 1.000000e-09 reg 6.000000e+05 train accuracy: 0.121367 val accuracy: 0.116000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.417571 val accuracy: 0.425000
lr 7.000000e-09 reg 5.000000e+02 train accuracy: 0.093265 val accuracy: 0.096000
lr 7.000000e-09 reg 5.000000e+04 train accuracy: 0.086918 val accuracy: 0.084000
lr 7.000000e-09 reg 5.000000e+05 train accuracy: 0.414469 val accuracy: 0.420000
lr 7.000000e-09 reg 6.000000e+05 train accuracy: 0.414510 val accuracy: 0.409000
lr 7.000000e-09 reg 5.000000e+06 train accuracy: 0.409265 val accuracy: 0.390000
lr 9.000000e-09 reg 5.000000e+02 train accuracy: 0.096755 val accuracy: 0.099000
lr 9.000000e-09 reg 5.000000e+04 train accuracy: 0.104898 val accuracy: 0.095000
lr 9.000000e-09 reg 5.000000e+05 train accuracy: 0.413347 val accuracy: 0.410000
lr 9.000000e-09 reg 6.000000e+05 train accuracy: 0.416327 val accuracy: 0.419000
lr 9.000000e-09 reg 5.000000e+06 train accuracy: 0.396816 val accuracy: 0.380000
lr 1.000000e-08 reg 5.000000e+02 train accuracy: 0.096694 val accuracy: 0.110000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.114061 val accuracy: 0.116000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.415347 val accuracy: 0.413000
lr 1.000000e-08 reg 6.000000e+05 train accuracy: 0.413408 val accuracy: 0.417000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.407388 val accuracy: 0.406000
lr 3.000000e-08 reg 5.000000e+02 train accuracy: 0.112571 val accuracy: 0.106000
lr 3.000000e-08 reg 5.000000e+04 train accuracy: 0.205653 val accuracy: 0.197000
lr 3.000000e-08 reg 5.000000e+05 train accuracy: 0.410020 val accuracy: 0.423000
lr 3.000000e-08 reg 6.000000e+05 train accuracy: 0.407163 val accuracy: 0.392000
lr 3.000000e-08 reg 5.000000e+06 train accuracy: 0.380714 val accuracy: 0.402000
best validation accuracy achieved: 0.425000
```

[6]:
```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)
```

```
0.412
```

[7]:
```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, it makes sense. For each image we are storing as a combination of :

1) (HOG) which implies texture of that image and

2)(color histogram) which accounts for the color variations of the image.

And this two features itself is not enough as HOG doesn't consider the variances due to translation and rotation.

For instance, the horse misclassifies many images due to presence of green color and for texture it just checks whether, each image have features such as eyes and ears. The color green is predominant for class 'deer'. And for class 'ship' , it tries to account the color of water, which ranges from blue to white. And for texture it tries to identify big blocky structures.

To solve more features should be added(eg: SIFT.).

### 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.

[8]:
```
# 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)
```

[9]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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

# net = TwoLayerNet(input_dim, hidden_dim, num_classes , reg =0.7)
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)*****
# Initialising data for Solver
data = {
    'X_train': X_train_feats,
    'X_val': X_val_feats,
    'X_test': X_test_feats,
    'y_train': y_train,
    'y_val': y_val,
    'y_test': y_test
}
```

```
best_val_accuracy = 0.0

results = {}
# Different learning and regularizations for cross validation
learning_rates = [7e-1 ,1e-2, 2.75e-2, 3e-1 ]
regularization_strengths = [ 1e-6 , 1e-4 , 5e-5]

# Storing all possible combinations of learning rates and regularization␣
↪constants
possible_combs = []
for i in learning_rates:
  for j in regularization_strengths:
    possible_combs.append((i,j))

# Iterating through each pair of the possibile_combs list
for l_rate, r_strength in possible_combs:
    solver = None
    # A two layer network is initialised with predefined values for each␣
↪parameter
    net = TwoLayerNet(input_dim, hidden_dim, num_classes , reg = r_strength)
    # Solver object created which does 10 epochs for each learning and␣
↪regularization
    # item and then the model is trained
    solver = Solver(net, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': l_rate,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                     verbose = False)
    solver.train()
    # The validation accuracy for each iteration is stored and compared with␣
↪already
    # existing values. And the network with the best accuracy is stored
    results[(l_rate , r_strength)] = solver.best_val_acc
    if solver.best_val_acc > best_val_accuracy:
        best_val_accuracy = solver.best_val_acc
        best_net = net

# For displaying the stored accuracies in an orderly manner
for lr, reg in sorted(results):
    val_accuracy = results[(lr, reg)]
    print('lr %e reg %e val accuracy: %f' % (
                lr, reg,  val_accuracy))
```

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

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

```
lr 1.000000e-02 reg 1.000000e-06 val accuracy: 0.482000
lr 1.000000e-02 reg 5.000000e-05 val accuracy: 0.484000
lr 1.000000e-02 reg 1.000000e-04 val accuracy: 0.477000
lr 2.750000e-02 reg 1.000000e-06 val accuracy: 0.527000
lr 2.750000e-02 reg 5.000000e-05 val accuracy: 0.529000
lr 2.750000e-02 reg 1.000000e-04 val accuracy: 0.534000
lr 3.000000e-01 reg 1.000000e-06 val accuracy: 0.595000
lr 3.000000e-01 reg 5.000000e-05 val accuracy: 0.586000
lr 3.000000e-01 reg 1.000000e-04 val accuracy: 0.592000
lr 7.000000e-01 reg 1.000000e-06 val accuracy: 0.577000
lr 7.000000e-01 reg 5.000000e-05 val accuracy: 0.569000
lr 7.000000e-01 reg 1.000000e-04 val accuracy: 0.566000
best validation accuracy achieved during cross-validation: 0.595000
```

[10]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

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

```
0.579
```