knn

October 6, 2024

[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 = 'Coursework/ENPM703/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/Coursework/ENPM703/assignment1/cs231n/datasets
/content/drive/My Drive/Coursework/ENPM703/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

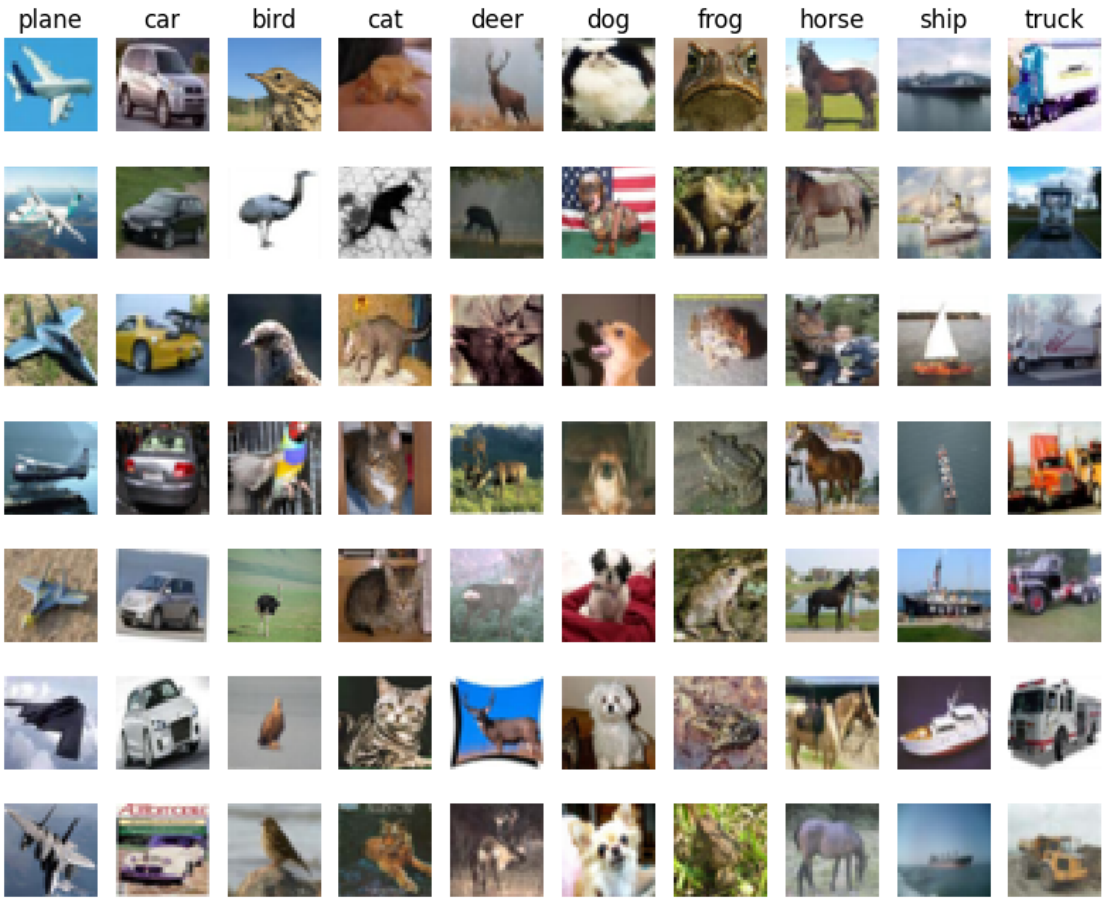
- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

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

[2]:
```
# 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
```

[3]:
```
# 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('````````````````````````````````````````````````')
```

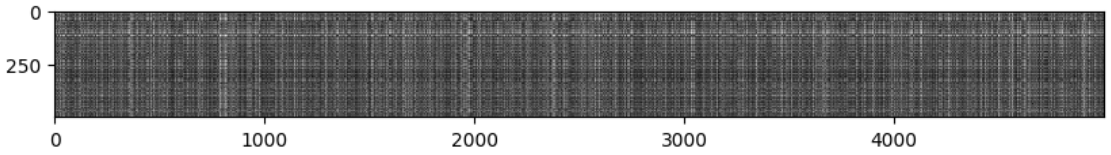
```
||||||||||||||||||||||||||||||||||||||||||||||||
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
``````````````````````````````````````````````````
```

[4]:
```
# 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('``')
```

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




```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

[5]:
```
# 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)
```

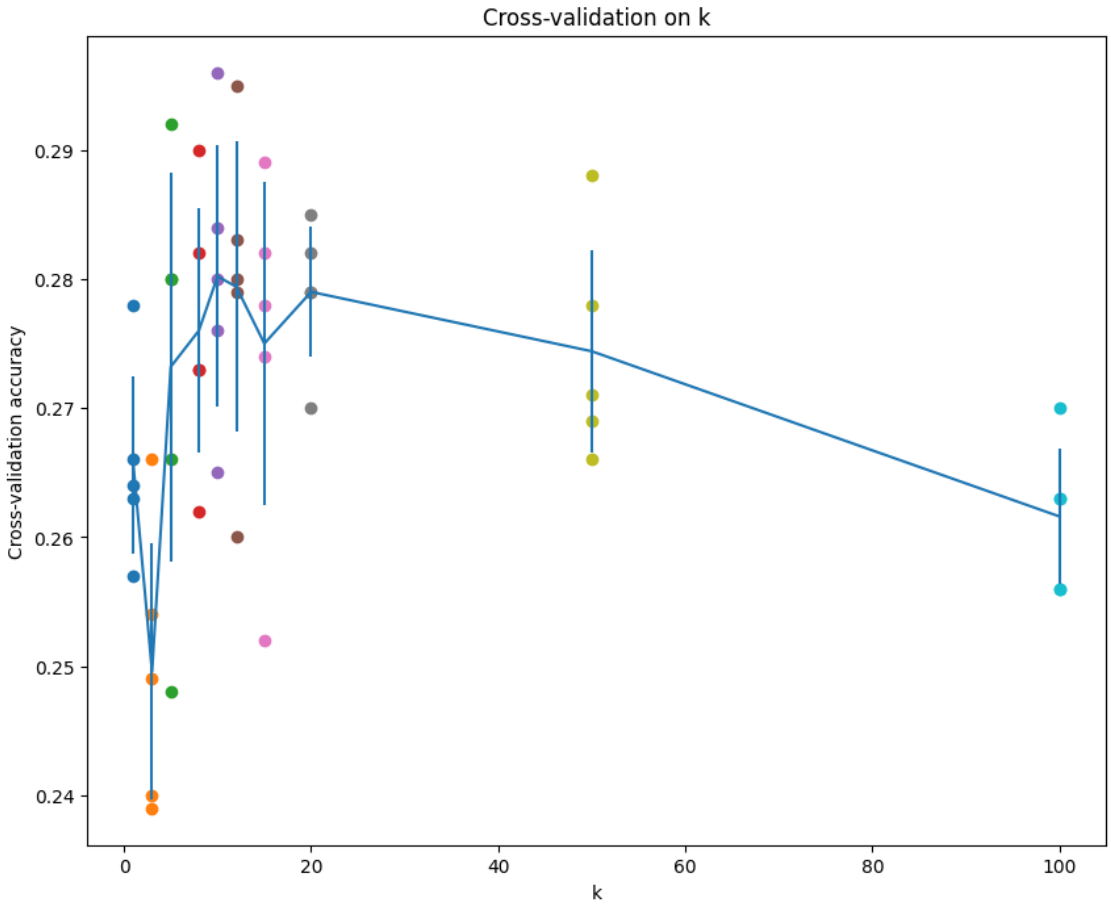
```
print('``````````````````````````````````````````````````')
```

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

[6]:
```
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.

[7]:
```
# 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)
```

```
```

[8]:
```
# 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('``````````````````````````````````````````````')
```

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




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

**Inline Question 1**

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

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

*Your Answer :*

Bright Rows: Outliers, indicate test images with high pixel-wise distances from most training images, suggesting they are outliers. This occurs when an image has distinct features, such as a unique background or content, leading to large pixel differences. For example, an image with a white background compared to others with dark backgrounds will result in a bright row.

Bright Columns: Similar Images, represent training images that are similar to many test images, often due to shared features like matching backgrounds or content. For instance, if several images have white backgrounds, a training image with a similar background will have smaller pixel differences, appearing in a bright column.

Conclusion- In summary, bright rows identify outliers, while bright columns highlight images that are more similar to the rest of the dataset. This analysis provides valuable insights into image comparisons and helps identify patterns in the data.*

[9]:
```
# 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`:

> Add blockquote

[10]:
```
# 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
````````````````````````````````````````````````
```

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 $\sigma$. 4. Subtracting the pixel-wise mean $\mu_{ij}$ and dividing by the pixel-wise standard deviation $\sigma_{ij}$. 5. Rotating the coordinate axes of the data.

*YourAnswer :*

**Subtracting the mean $\mu$ Subtracting the per-pixel mean $\mu_{ij}$:**

*YourExplanation :*

When preparing data for a Nearest Neighbor classifier that uses L1 distance, some preprocessing steps will change the performance while others won't. Here's the reasoning for each step:

1. **Subtracting the mean $\mu$**:
   This step shifts all pixel values by a constant (the global mean), but it doesn't affect the relative differences between images. L1 distance depends on absolute differences, so subtracting the global mean won't change the classifier's performance. **This step does not change performance**.

2. **Subtracting the per-pixel mean $\mu_{ij}$**:
   This normalizes each pixel value by its own mean across images, which still maintains the relative differences between images at each pixel location. Therefore, the L1 distances between images stay the same, meaning the classifier's performance is unaffected. **This step does not change performance**.

3. **Subtracting the mean $\mu$ and dividing by the standard deviation $\sigma$**:
   Dividing by the standard deviation scales the entire dataset. Although this doesn't change the ranking of distances, it changes the magnitude of L1 distances, which could affect the classification threshold and performance. **This step will change performance**.

4. **Subtracting the pixel-wise mean $\mu_{ij}$ and dividing by the pixel-wise standard deviation $\sigma_{ij}$**:
   Normalizing pixel values individually (both subtracting the mean and dividing by the standard deviation) adjusts the data significantly, changing the relative distances between images. **This step will change performance**.

5. **Rotating the coordinate axes of the data**:
   L1 distance is sensitive to the specific pixel locations. Rotating the data changes which pixels are compared, altering the L1 distances and thus affecting performance. **This step will change performance**.

[11]:
```
# 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
``````````````````````````````````````````````````
```

[12]:
```
# 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
``````````````````````````````````````````````````
```

[13]:
```
# 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 38.592718 seconds
One loop version took 47.866453 seconds
No loop version took 0.423912 seconds
``````````````````````````````````````````````
```

### 1.0.1 Cross-validation

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

[14]:
```
# 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)*****

# Split the training data into folds
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
# pass

# *****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)*****
# pass

# Perform k-fold cross-validation
# Iterate over each candidate value for k
for k in k_choices:
    # Create an empty list to hold accuracy results for the current k
    k_to_accuracies[k] = []

    # Loop through each fold in the cross-validation process
    for fold in range(num_folds):
        # Initialize a new k-NN classifier for this fold
        knn = KNearestNeighbor()

        # Combine all training folds except the current validation fold
        X_train_combined = np.concatenate(X_train_folds[:fold] +␣
 ↪X_train_folds[fold + 1:])
        y_train_combined = np.concatenate(y_train_folds[:fold] +␣
 ↪y_train_folds[fold + 1:])

        # Train the k-NN classifier on the combined training data
        knn.train(X_train_combined, y_train_combined)
```

```
        # Predict labels for the current validation fold
        y_pred = knn.predict(X_train_folds[fold], k=k)

        # Compute accuracy and add it to the list for this k
        accuracy = np.mean(y_pred == y_train_folds[fold])
        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 = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
```

```
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```````````````````````````````````````````````````

[15]:
```
# 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('`````````````````````````````````````````````````')
```

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




```
```

[16]:
```
# 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[accuracies_mean.argmax()]
print('The best k is ', best_k)

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

```
||||||||||||||||||||||||||||||||||||||||||||||
The best k is  10
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.

*YourAnswer* : 2,4

*YourExplanation* : 1. The decision boundary of the k-NN classifier is linear.

False. The decision boundary of the k-NN classifier is not linear. If we consider a dataset where classes belong to concentric circles, the decision boundaries will follow the curvature of the concentric circles, making them non-linear.

2. The training error of a 1-NN will always be lower than that of 5-NN.

   True. The training error of a 1-NN will always be lower than that of 5-NN because for each training example, its nearest neighbor is always itself, resulting in zero error.

3. The test error of a 1-NN will always be lower than that of a 5-NN.

   False. The test error of a 1-NN will not always be lower than 5-NN. The 1-NN might misclassify it with high probability (e.g., 100%), while the 5-NN might correctly classify it with high probability (e.g., 100%). The value of k is data-dependent, and we need to perform cross-validation to determine the best k for our dataset.

4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.

True. The testing phase of k-NN requires comparisons of each test sample with the entire training set, which grows with the size of the training set. However, we can use Approximate Nearest Neighbor techniques (like k-d trees or ball trees) to improve time complexity. Additionally, the training phase of k-NN, which involves remembering the training set, also grows with the size of the training set.
```

svm

October 6, 2024

[ ]:
```
# 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 = 'Coursework/ENPM703/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/Coursework/ENPM703/assignment1/cs231n/datasets
/content/drive/My Drive/Coursework/ENPM703/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
```

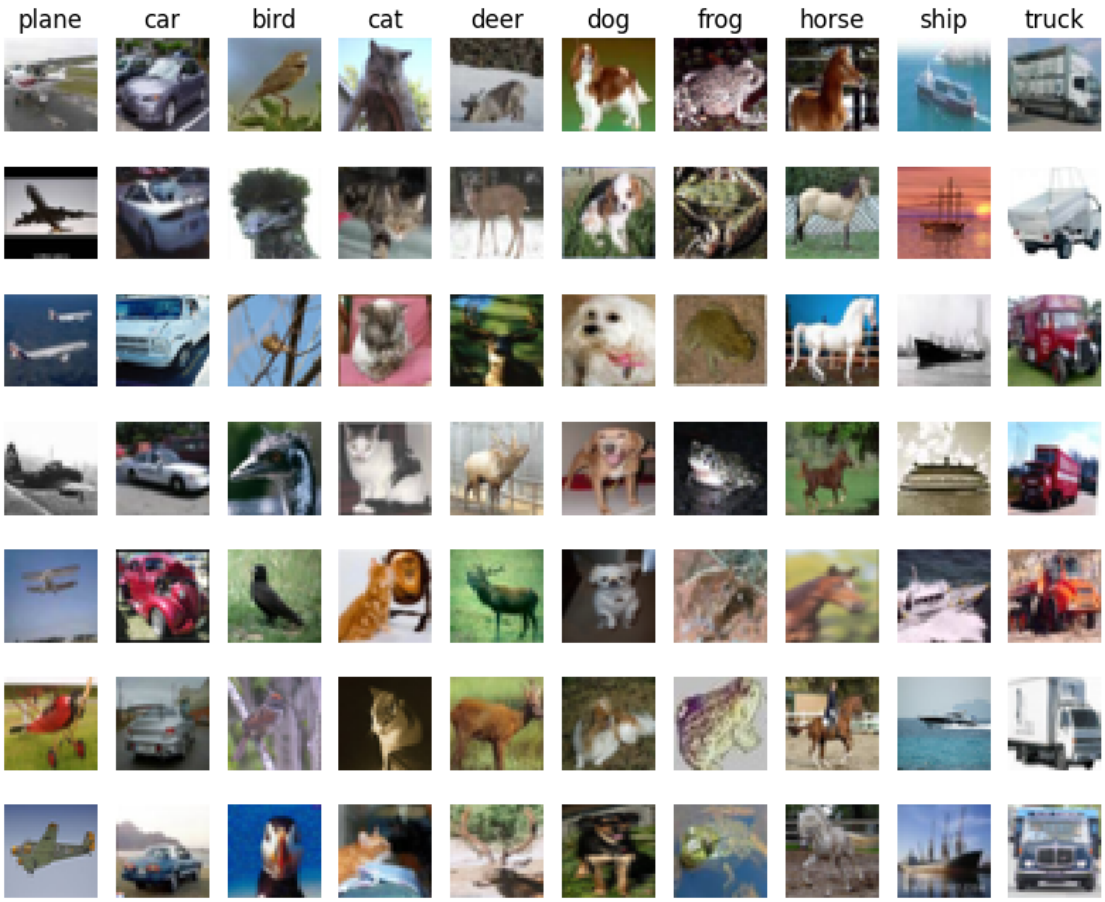
1.1 CIFAR-10 Data Loading and Preprocessing

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

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



[ ]:
```
# 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)
```

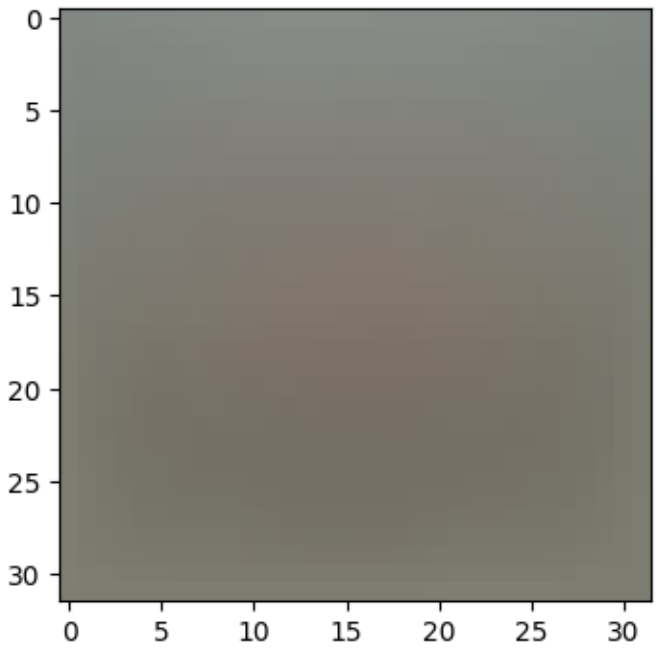
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[ ]:
```
# 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: 8.548761
```

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: -5.567280 analytic: -5.567280, relative error: 3.455474e-11
numerical: 17.747856 analytic: 17.747856, relative error: 1.093367e-12
numerical: 16.790937 analytic: 16.790937, relative error: 2.579964e-12
numerical: -5.564738 analytic: -5.564738, relative error: 2.006763e-11
numerical: -17.339474 analytic: -17.339474, relative error: 3.969571e-12
numerical: 38.907350 analytic: 38.907350, relative error: 9.012509e-13
numerical: -17.296095 analytic: -17.296095, relative error: 1.838287e-11
numerical: 0.873550 analytic: 0.873550, relative error: 3.372058e-10
numerical: 9.632719 analytic: 9.632719, relative error: 1.883329e-11
numerical: -3.981775 analytic: -3.981775, relative error: 1.166418e-10
numerical: -11.515117 analytic: -11.515117, relative error: 9.292111e-12
numerical: 21.201675 analytic: 21.201675, relative error: 7.863947e-13
numerical: 14.210786 analytic: 14.210786, relative error: 9.292920e-12
numerical: 22.311410 analytic: 22.311410, relative error: 9.129945e-12
numerical: 29.844449 analytic: 29.844449, relative error: 8.701303e-13
numerical: -14.388139 analytic: -14.388139, relative error: 4.559185e-12
numerical: -8.230390 analytic: -8.230390, relative error: 6.251551e-11
numerical: 11.404340 analytic: 11.404340, relative error: 4.996906e-12
numerical: -9.771559 analytic: -9.730924, relative error: 2.083620e-03
numerical: 47.267472 analytic: 47.267472, relative error: 1.841016e-12
```

Inline Question 1

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

YourAnswer :

Since the loss function is not differentiable at some locations, gradient check inconsistencies in SVM can happen. The SVM loss function, which is specified as max(0, margin), is non-differentiable at this point because it has a "kink" where the margin equals zero.

These random inconsistencies don't pose a big threat. In the differentiable parts of the function, which make up most of the space, gradient descent can still be carried out efficiently. It is uncommon for the loss to occur precisely on these non-differentiable places in real life.

Example

A straightforward instance in one dimension where a gradient check may not succeed is the function $f(x) = |x|$ at $x = 0$. Similar to the SVM loss, this function has an undefined gradient at zero, which causes it to kink. The probability of running into these non-differentiable locations varies with the margin. The SVM loss function's challenging region, when margin = 0, is pushed away from the function by increasing the margin (delta). This lowers the probability of training at non-differentiable sites, which lowers the frequency of gradient check mismatches.

These possible inconsistencies stem from the non-strict differentiability of the SVM loss function, as suggested. The loss function's max operation, which produces locations where the gradient is mathematically undetermined, has this property by default.

[ ]:
```
# 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: 8.548761e+00 computed in 0.215453s
Vectorized loss: 8.548761e+00 computed in 0.016615s
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 0.117929s
Vectorized loss and gradient: computed in 0.009809s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

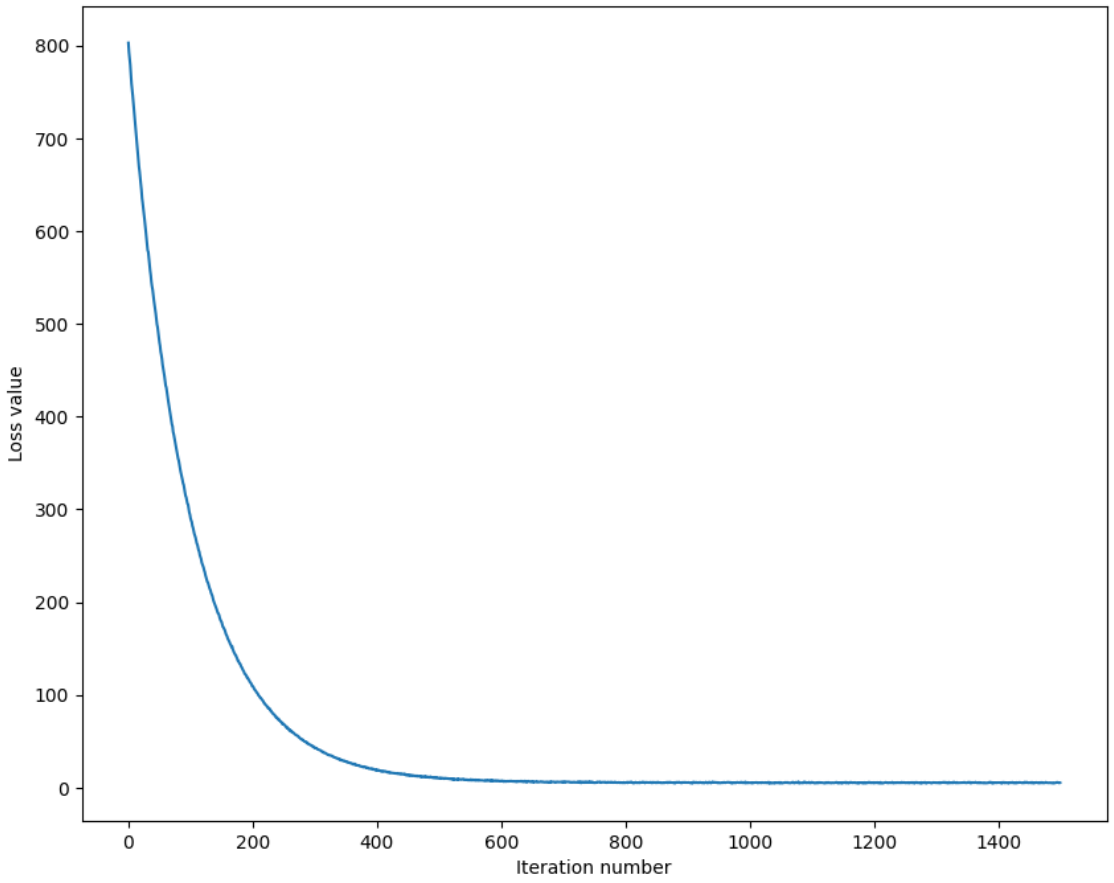
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[ ]:
```
# 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 802.779880
iteration 100 / 1500: loss 290.328074
iteration 200 / 1500: loss 109.306737
iteration 300 / 1500: loss 42.505855
iteration 400 / 1500: loss 19.213527
iteration 500 / 1500: loss 9.990988
iteration 600 / 1500: loss 7.411950
iteration 700 / 1500: loss 6.398814
iteration 800 / 1500: loss 5.393731
iteration 900 / 1500: loss 5.497243
iteration 1000 / 1500: loss 5.373747
iteration 1100 / 1500: loss 5.644619
iteration 1200 / 1500: loss 5.625448
```

```
iteration 1300 / 1500: loss 5.218067
iteration 1400 / 1500: loss 5.483539
That took 10.257177s
```

[ ]:
```
# 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: 0.370571
validation accuracy: 0.377000
```

[ ]:
```
# 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-3]
regularization_strengths = [2.5e4, 5e4, 1e4,3e4]
# hyperparamters values to do grid search


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

# Create a grid of hyperparameter combinations using learning rates and␣
 ↪regularization strengths
grid_search = [(lr, reg) for lr in learning_rates for reg in␣
 ↪regularization_strengths]

# Iterate through each hyperparameter configuration
```

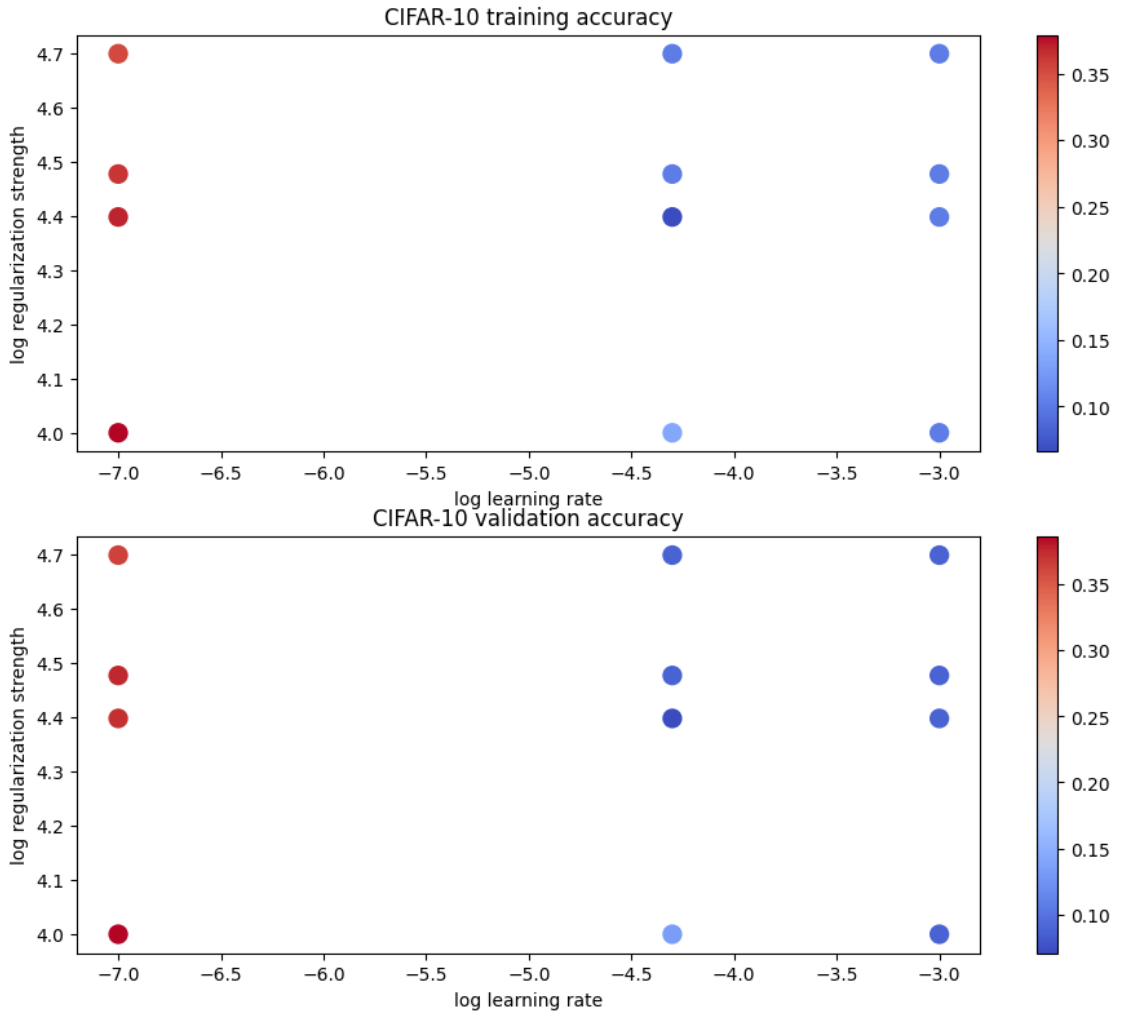
```
for config_index, (lr, reg) in enumerate(grid_search):
    print(f"Evaluating Hyperparameter Configuration #{config_index + 1} of␣
 ↪{len(grid_search)}")
    print(f"Current Configuration: Learning Rate = {lr}, Regularization␣
 ↪Strength = {reg}")

    svm_model = LinearSVM()

    # Train the SVM model on the training dataset
    train_loss = svm_model.train(X_train, y_train, learning_rate=lr,
                                 reg=reg, num_iters=1500, verbose=False)

    # Generate predictions for the training and validation datasets
    y_train_pred = svm_model.predict(X_train)
    y_val_pred = svm_model.predict(X_val)

    # Calculate the accuracy for both datasets
    train_accuracy = np.mean(y_train_pred == y_train)
    val_accuracy = np.mean(y_val_pred == y_val)

    # Record the results for this configuration
    results[(lr, reg)] = (train_accuracy, val_accuracy)

    # Update the best validation accuracy and the corresponding SVM model if␣
 ↪applicable
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_svm = svm_model


# pass

# *****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)
```

```
Evaluating Hyperparameter Configuration #1 of 12
Current Configuration: Learning Rate = 1e-07, Regularization Strength = 25000.0
Evaluating Hyperparameter Configuration #2 of 12
Current Configuration: Learning Rate = 1e-07, Regularization Strength = 50000.0
```

```
Evaluating Hyperparameter Configuration #3 of 12
Current Configuration: Learning Rate = 1e-07, Regularization Strength = 10000.0
Evaluating Hyperparameter Configuration #4 of 12
Current Configuration: Learning Rate = 1e-07, Regularization Strength = 30000.0
Evaluating Hyperparameter Configuration #5 of 12
Current Configuration: Learning Rate = 5e-05, Regularization Strength = 25000.0

/content/drive/My
Drive/Coursework/ENPM703/assignment1/cs231n/classifiers/linear_svm.py:123:
RuntimeWarning: overflow encountered in scalar multiply
  loss += reg * np.sum(W * W)
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:88:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/drive/My
Drive/Coursework/ENPM703/assignment1/cs231n/classifiers/linear_svm.py:123:
RuntimeWarning: overflow encountered in multiply
  loss += reg * np.sum(W * W)

Evaluating Hyperparameter Configuration #6 of 12
Current Configuration: Learning Rate = 5e-05, Regularization Strength = 50000.0

/content/drive/My
Drive/Coursework/ENPM703/assignment1/cs231n/classifiers/linear_svm.py:155:
RuntimeWarning: overflow encountered in multiply
  dW += 2 * reg * W  # Apply regularization to dW
/content/drive/My
Drive/Coursework/ENPM703/assignment1/cs231n/classifiers/linear_classifier.py:82:
RuntimeWarning: invalid value encountered in subtract
  self.W -= learning_rate * gradient

Evaluating Hyperparameter Configuration #7 of 12
Current Configuration: Learning Rate = 5e-05, Regularization Strength = 10000.0
Evaluating Hyperparameter Configuration #8 of 12
Current Configuration: Learning Rate = 5e-05, Regularization Strength = 30000.0

/content/drive/My
Drive/Coursework/ENPM703/assignment1/cs231n/classifiers/linear_svm.py:111:
RuntimeWarning: overflow encountered in subtract
  loss_margins = np.maximum(0, scores_matrix - true_class_scores + 1)  # Delta =
1
/content/drive/My
Drive/Coursework/ENPM703/assignment1/cs231n/classifiers/linear_svm.py:111:
RuntimeWarning: invalid value encountered in subtract
  loss_margins = np.maximum(0, scores_matrix - true_class_scores + 1)  # Delta =
1

Evaluating Hyperparameter Configuration #9 of 12
Current Configuration: Learning Rate = 0.001, Regularization Strength = 25000.0
Evaluating Hyperparameter Configuration #10 of 12
Current Configuration: Learning Rate = 0.001, Regularization Strength = 50000.0
```

```
Evaluating Hyperparameter Configuration #11 of 12
Current Configuration: Learning Rate = 0.001, Regularization Strength = 10000.0
Evaluating Hyperparameter Configuration #12 of 12
Current Configuration: Learning Rate = 0.001, Regularization Strength = 30000.0
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.379204 val accuracy: 0.386000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.369469 val accuracy: 0.372000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.362673 val accuracy: 0.376000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.353143 val accuracy: 0.363000
lr 5.000000e-05 reg 1.000000e+04 train accuracy: 0.136490 val accuracy: 0.133000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.065918 val accuracy: 0.071000
lr 5.000000e-05 reg 3.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 3.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.386000
```

[ ]:
```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()
```

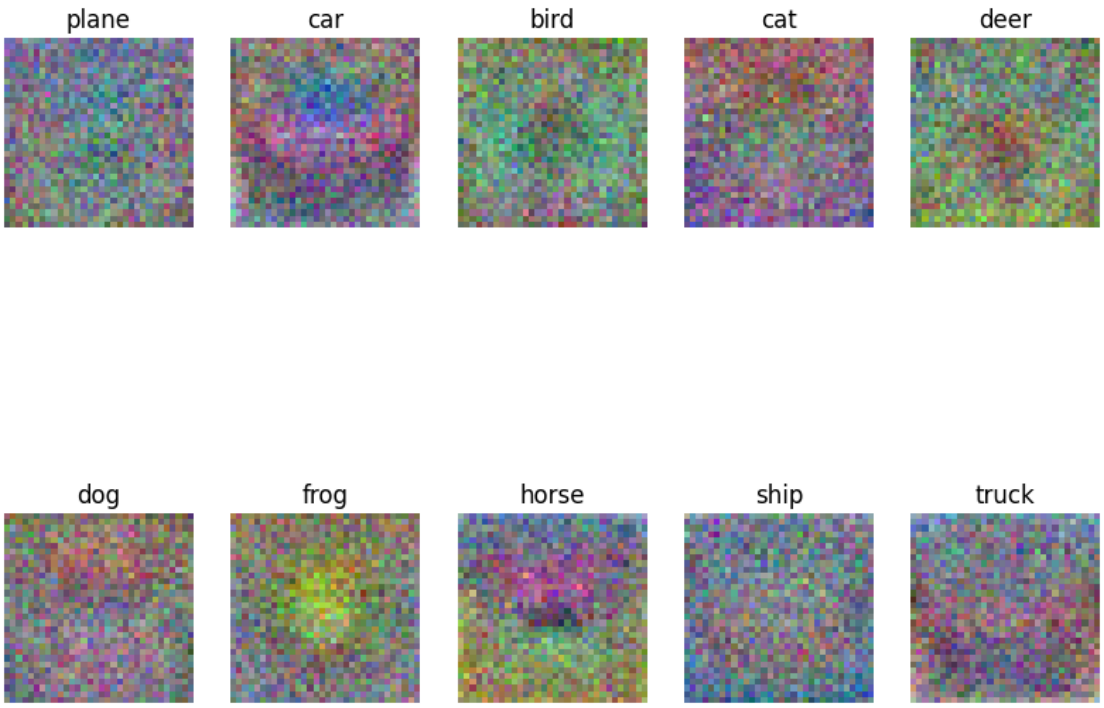


[ ]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.390000
```

[ ]:
```
# 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.

YourAnswer : The SVM weights that are displayed visually stand in for the templates that each class has acquired through data analysis. The "essential construction" of the training images that are part of a specific class is essentially described by each of them.The images are blurred because the accuracy obtained was very low. For example, because the dataset probably contains photos of horses with some of them looking left and some looking right, the weights of the class "horse" resemble a horse with two heads. To forecast the class of a given test sample, we use k-NN to compare a test picture with all of the training examples using a suitable distance measure (such L1 or L2). With SVM, on the other hand, we compare the test image with the templates of each class by utilizing the inner product.

softmax

October 6, 2024

[ ]:
```
# 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 = 'Coursework/ENPM703/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/Coursework/ENPM703/assignment1/cs231n/datasets
/content/drive/My Drive/Coursework/ENPM703/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: 2.392091
sanity check: 2.302585
```

Inline Question 1

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

Your Answer :

The weights in our model are initialized at random at the beginning of training. Prior to any learning taking place, all classes are deemed equally likely to be selected when random initialization is used. There are ten classes in the CIFAR-10 dataset. As a result, the initial chance of selecting the right class is 1/10, or 0.1. The negative log probability of the correct class is computed via the softmax loss function. Therefore, the predicted initial loss becomes -log(0.1) assuming the equal probability across all classes. The situation where our model is essentially generating random guesses and has no learned preference for any class is represented by this initial loss number. It is crucial to remember that this is only the beginning. We anticipate that this loss will go down as the training process goes on and the model gains knowledge from the data, showing increased classification accuracy. The model's progress during training can be gauged by comparing it to the -log(0.1) baseline.

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

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

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

```
numerical: 1.377989 analytic: 1.377989, relative error: 1.396144e-08
numerical: 2.684691 analytic: 2.684690, relative error: 3.455048e-08
numerical: 1.528461 analytic: 1.528461, relative error: 5.988974e-08
numerical: 1.208482 analytic: 1.208482, relative error: 3.937451e-08
numerical: -0.858643 analytic: -0.858643, relative error: 3.519605e-08
numerical: 3.047423 analytic: 3.047423, relative error: 8.374732e-09
numerical: 1.955854 analytic: 1.955854, relative error: 1.372885e-09
numerical: -1.738256 analytic: -1.738256, relative error: 1.779607e-08
numerical: 4.291872 analytic: 4.291872, relative error: 6.053295e-09
numerical: -0.388025 analytic: -0.388025, relative error: 1.245113e-08
numerical: 2.243866 analytic: 2.243866, relative error: 2.659617e-09
numerical: 1.773066 analytic: 1.773066, relative error: 8.770800e-09
numerical: 0.264935 analytic: 0.264935, relative error: 1.634920e-07
numerical: -1.670226 analytic: -1.670226, relative error: 9.073044e-09
numerical: -0.098163 analytic: -0.098163, relative error: 7.411439e-07
numerical: 2.348323 analytic: 2.348323, relative error: 1.330503e-08
numerical: -0.883782 analytic: -0.883782, relative error: 4.716421e-08
numerical: 0.752223 analytic: 0.752223, relative error: 4.796779e-08
numerical: -3.088259 analytic: -3.088259, relative error: 1.208118e-08
numerical: 1.033050 analytic: 1.033050, relative error: 2.029387e-08
```

[ ]:
```
# 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: 2.392091e+00 computed in 0.019746s
vectorized loss: 2.392091e+00 computed in 0.013107s
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, 2e-6, 2.5e-6]
regularization_strengths = [1e3, 1e4, 2e4, 2.5e4, 3e4, 3.5e4]

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

# pass

# Generate combinations of learning rates and regularization strengths for grid␣
 ↪search
grid_search = [(lr, reg) for lr in learning_rates for reg in␣
 ↪regularization_strengths]

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

for learning_rate, regularization_strength in grid_search:
    softmax = Softmax()

    # Train the Softmax classifier on the training set
    loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate,
```

```
                              reg=regularization_strength, num_iters=1500,␣
 ↪verbose=False)

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

    # Calculate accuracy for training and validation sets
    train_accuracy = np.mean(y_train_pred == y_train)
    val_accuracy = np.mean(y_val_pred == y_val)

    # Store results
    results[(learning_rate, regularization_strength)] = (train_accuracy,␣
 ↪val_accuracy)

    # Update best validation accuracy and corresponding Softmax model
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_softmax = softmax

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

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

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

```
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.267531 val accuracy: 0.281000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.356327 val accuracy: 0.367000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.337633 val accuracy: 0.350000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.324776 val accuracy: 0.342000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.324898 val accuracy: 0.336000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.313000 val accuracy: 0.332000
lr 2.000000e-06 reg 1.000000e+03 train accuracy: 0.392673 val accuracy: 0.383000
lr 2.000000e-06 reg 1.000000e+04 train accuracy: 0.336837 val accuracy: 0.335000
lr 2.000000e-06 reg 2.000000e+04 train accuracy: 0.328347 val accuracy: 0.342000
lr 2.000000e-06 reg 2.500000e+04 train accuracy: 0.317980 val accuracy: 0.347000
lr 2.000000e-06 reg 3.000000e+04 train accuracy: 0.301122 val accuracy: 0.322000
lr 2.000000e-06 reg 3.500000e+04 train accuracy: 0.287857 val accuracy: 0.289000
lr 2.500000e-06 reg 1.000000e+03 train accuracy: 0.392204 val accuracy: 0.391000
lr 2.500000e-06 reg 1.000000e+04 train accuracy: 0.347490 val accuracy: 0.359000
lr 2.500000e-06 reg 2.000000e+04 train accuracy: 0.303122 val accuracy: 0.300000
lr 2.500000e-06 reg 2.500000e+04 train accuracy: 0.306755 val accuracy: 0.314000
```

```
lr 2.500000e-06 reg 3.000000e+04 train accuracy: 0.299347 val accuracy: 0.303000
lr 2.500000e-06 reg 3.500000e+04 train accuracy: 0.281245 val accuracy: 0.289000
best validation accuracy achieved during cross-validation: 0.391000
```

[ ]:
```
# 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: 0.377000
```

Inline Question 2 - *True or False*

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

YourAnswer : True

YourExplanation :

SVM Loss: The SVM loss is a local objective whose sole concern is preserving a given margin between the scores of the right and wrong classes. The SVM loss for a new datapoint will be 0 if it is added and the correct class score is at least the margin greater than the incorrect class scores. For instance, if the right class was 1, and the margin was 2, and the scores were [10, 8, 7], the SVM loss would be: max(0, 8 + 2 - 10) + max(0, 7 + 2 - 10) = 0. The total SVM loss would remain unchanged if this datapoint were added.

Softmax Classifier Loss:

In contrast, the Softmax classifier takes into account each individual score while determining its loss. Since adding a new datapoint alters the normalization of scores across all classes, it always has an impact on the Softmax loss. The Softmax loss in the same scenario with scores [10, 8, 7] would be:

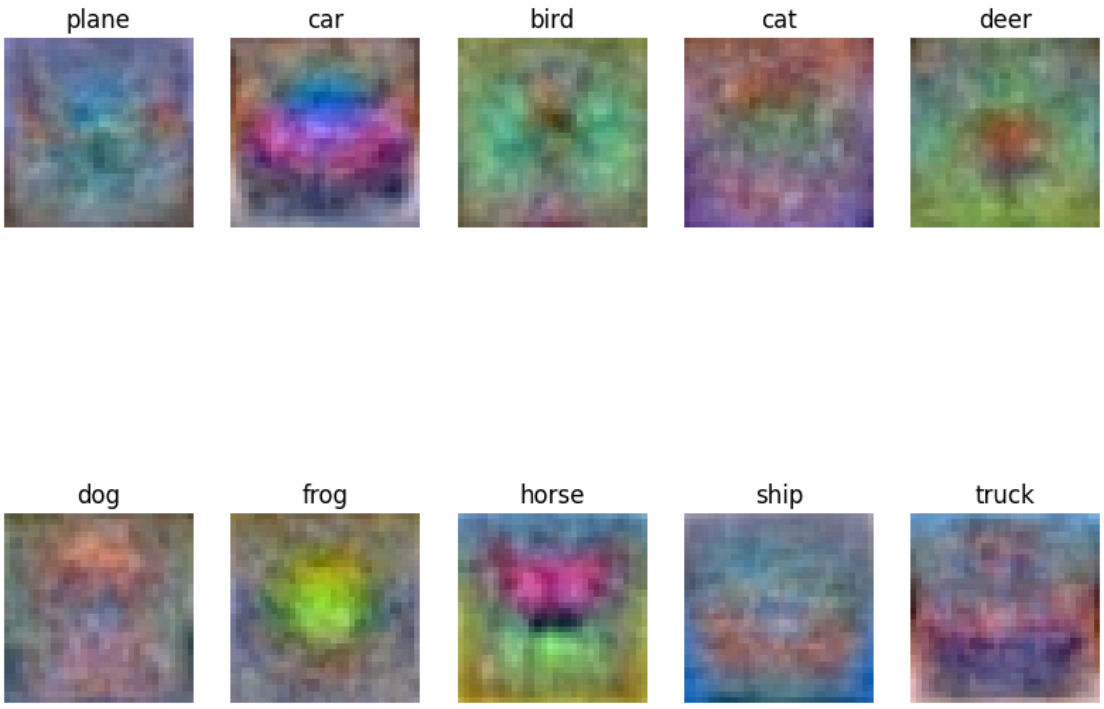
-log(softmax(10)) -log(0.84) 0.17

The dataset's overall Softmax loss would rise as a result of this non-zero loss.

The key difference is that the Softmax loss is sensitive to the precise values of each score, but the SVM loss is dependent on the threshold effect (the margin). This enables the addition of datapoints that consistently impact the Softmax loss but have no effect on the SVM loss.

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

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
for i in range(10):
```

```
    plt.subplot(2, 5, i + 1)

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



[ ]:

two_layer_net

October 6, 2024

[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 = 'Coursework/ENPM703/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/Coursework/ENPM703/assignment1/cs231n/datasets
/content/drive/My Drive/Coursework/ENPM703/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))))
```

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

[ ]:
```
# 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:  9.769848888397517e-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)
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))
```

```
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 vanishing gradient problem arises during backpropagation when gradient flow is close to zero.

1. The sigmoid function has a vanishing gradient problem, meaning that for very large positive and negative input values (in the tail regions of the curve), the gradient is almost zero. Saturation can occur in one dimension when very large positive and negative numbers, such as [-1e3, 1e3], are considered.
2. ReLU's linear response to a positive input makes it less susceptible to the vanishing gradient problem, which is one of its main advantages over Sigmoid. The gradient of ReLU is either 1 for positive inputs or 0 for negative inputs. The vanishing gradient problem may affect ReLU

in the unlikely event that all of the input values are negative. Certain neurons are unable to learn more after this. The term "dying ReLU problem" refers to this. One way to approach the vanishing gradient problem in one dimension is to focus solely on negative numbers, such as [-1, -2, -3].

3. Leaky ReLU applies a slight negative slope for negative values, i.e., if x $<$ 0 then 0.01x else x, in an attempt to overcome the ReLU problem of "dead" neurons. Thus, the goal of Leaky ReLU is to resolve the vanishing gradient issue. The gradient at x = 0 is undefined, though, because the max(0.01x, x) function is not continuous at that point. Therefore, if it is not explicitly handled in code, considering all zero values, such as [0, 0, 0], is a one-dimensional example that can result in zero gradients and can only occur due to a poor network setup.

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 sum_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: 1.22e-08
W2 relative error: 3.28e-10
b1 relative error: 8.37e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-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)*****

# pass
# Initialize and train the Solver with the specified model, data, and training␣
 ↪parameters
solver = Solver(model, data, update_rule="sgd",
                optim_config={"learning_rate": 1e-4}, lr_decay=0.95,
                batch_size=230, print_every=100, num_epochs=5)
solver.train()
```

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

```
(Iteration 1 / 1065) loss: 2.301819
(Epoch 0 / 5) train acc: 0.148000; val_acc: 0.140000
(Iteration 101 / 1065) loss: 2.256008
(Iteration 201 / 1065) loss: 2.124213
(Epoch 1 / 5) train acc: 0.247000; val_acc: 0.241000
(Iteration 301 / 1065) loss: 2.105422
(Iteration 401 / 1065) loss: 2.020780
(Epoch 2 / 5) train acc: 0.278000; val_acc: 0.292000
(Iteration 501 / 1065) loss: 1.916513
(Iteration 601 / 1065) loss: 1.938223
(Epoch 3 / 5) train acc: 0.324000; val_acc: 0.316000
(Iteration 701 / 1065) loss: 1.908146
(Iteration 801 / 1065) loss: 1.879034
(Epoch 4 / 5) train acc: 0.342000; val_acc: 0.331000
(Iteration 901 / 1065) loss: 1.810761
(Iteration 1001 / 1065) loss: 1.765554
(Epoch 5 / 5) train acc: 0.341000; val_acc: 0.358000
```

10 Debug the training

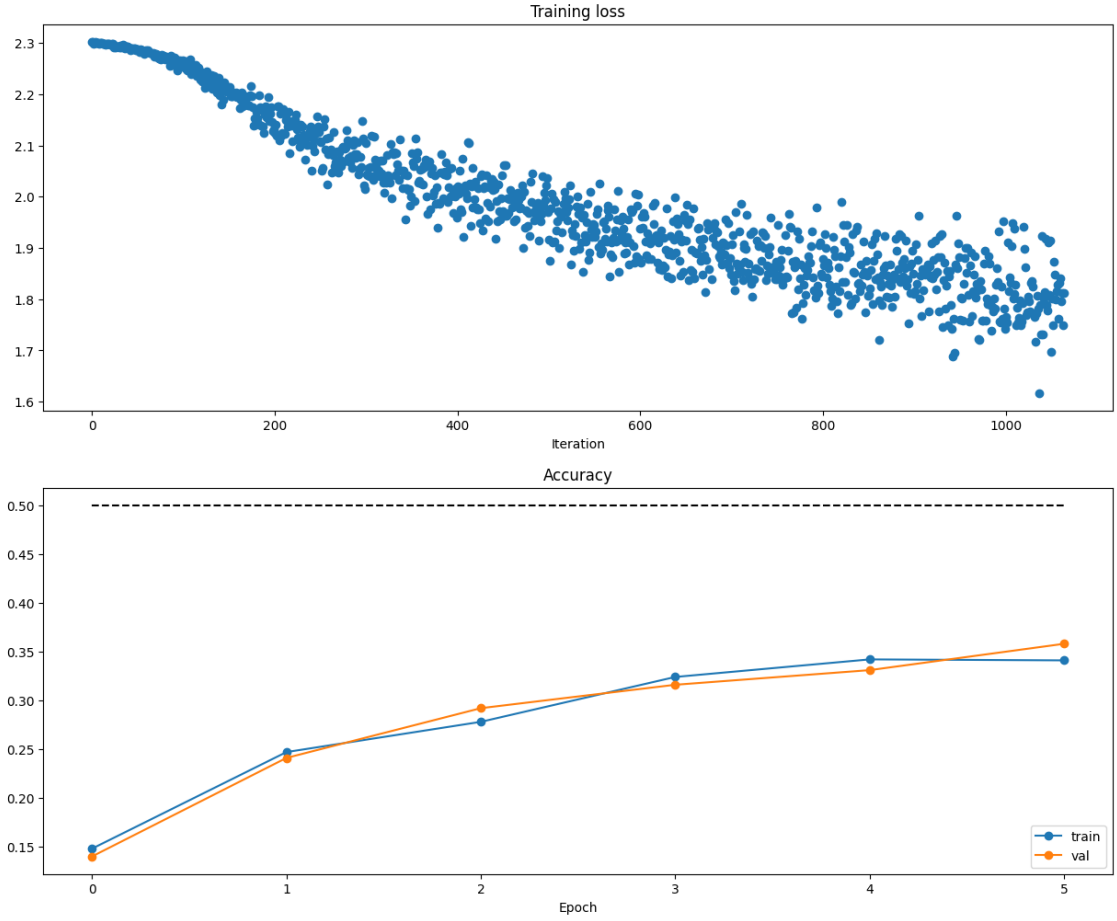
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

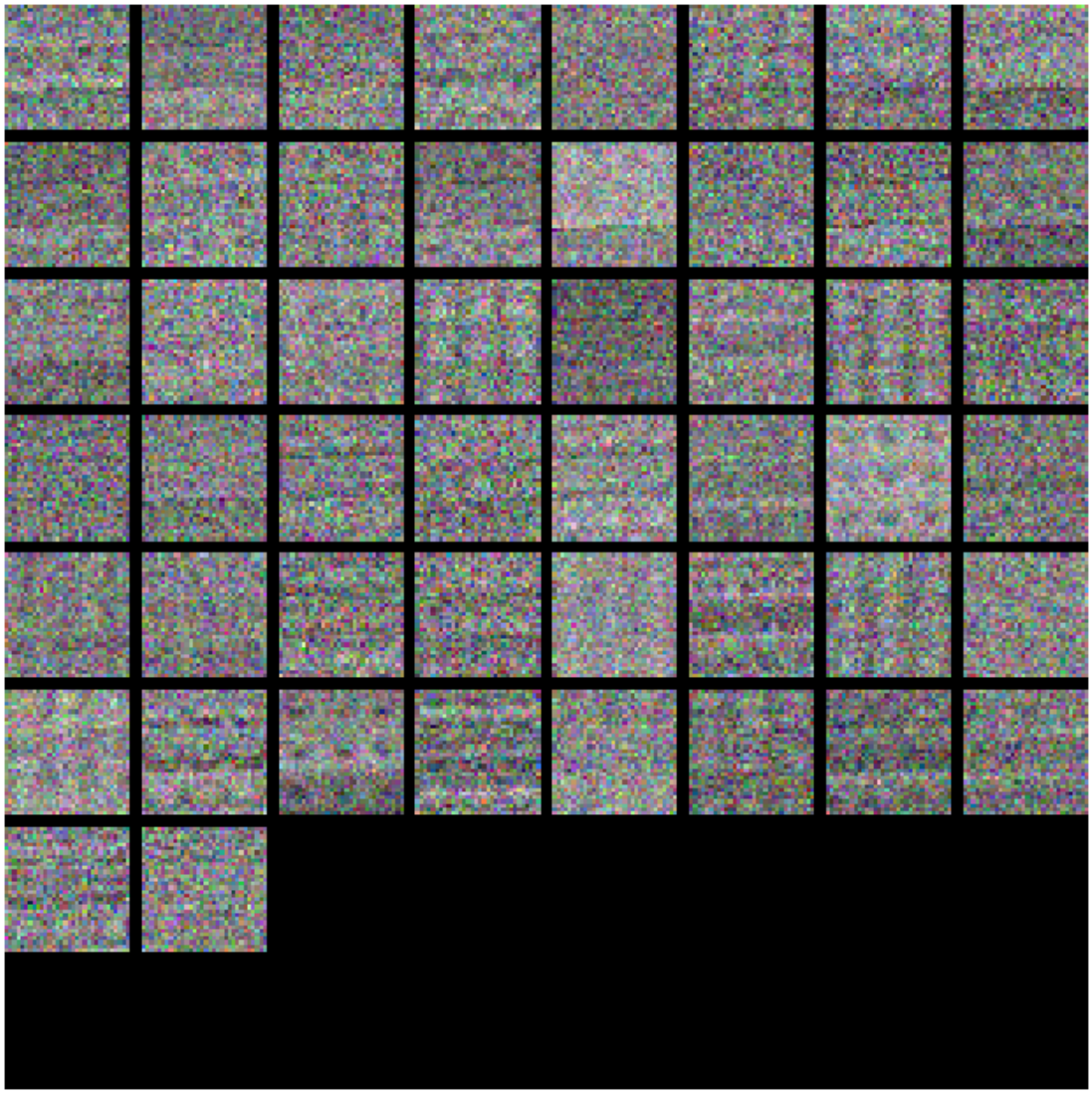
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[ ]:
```
# 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.).

[ ]:
```
model = TwoLayerNet()
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                          #
################################################################################
# ****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)****

#given the min-max ranges of the hyperparameters, create random combinations of␣
 ↪the parameters
# with a probability distribution that is uniform
def generate_random_hyperparams(lr_range, reg_range, hidden_size_range,␣
 ↪epoch_values):
    """ Generate arbitrary hyperparameters for the number of epochs, hidden␣
 ↪layer size, regularization intensity, and learning rate.

    Args:
    - lr_range (tuple): The learning rate exponent range, for example, (-4, -2).
    - reg_range (tuple): The regularization strength exponent range, for␣
 ↪example, (-7, -3).
    - hidden_size_range (tuple): The range, such as (50, 200), for the size of␣
 ↪the hidden layer.
    - epoch_values (list): A list of potential values for an epoch, such as␣
 ↪[10, 20].

    Returns: - (float, float, int, int): A tuple with the following contents:␣
 ↪epochs, regularization strength, hidden size, and randomly sampled learning␣
 ↪rate
    """
    lr = 10**np.random.uniform(lr_range[0], lr_range[1])
    reg = 10**np.random.uniform(reg_range[0], reg_range[1])
    hidden_size = np.random.randint(hidden_size_range[0], hidden_size_range[1])
    epochs = np.random.choice(epoch_values)

    return lr, reg, hidden_size, epochs
```

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

# form random combinations of hyperparameters
grid_search = [generate_random_hyperparams((-2, -4), (-7, -3), (50, 100, 200),␣
 ↪(10, 20))
               for count in range(num_hyperparam_configs)]

# get our data
# data = get_CIFAR10_data()

best_val = -1
results = {}

# iterate over the generated hyperparameter configurations
for config_num, config in enumerate(grid_search):
    print("Hyperparam config #{} of #{}: ".format(config_num + 1,␣
 ↪len(grid_search)), end='')

    lr, reg, hidden_size, epochs = config
    print("lr: {:.2e}, reg: {:.2e}, hidden_size: {}, epochs: {}".format(lr,␣
 ↪reg, hidden_size, epochs))

    # initialize the model with the random hyperparameters
    model = TwoLayerNet(hidden_dim=hidden_size, reg=reg)
    current_solver = Solver(model, data, update_rule='sgd',␣
 ↪optim_config={'learning_rate': lr},
                            lr_decay=0.95, num_epochs=epochs, batch_size=100,
                            print_every=100, verbose=False)

    # train amodel
    current_solver.train()

    # store the best val accuracy and the TwoLayerNet object
    if current_solver.best_val_acc > best_val:
        best_val = current_solver.best_val_acc
        solver = current_solver

    # print results
    print('Validation accuracy: %.4f' % (solver.best_val_acc,))
    print()

print('Best validation accuracy achieved: %.4f' % best_val)

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

```
################################################################################
```

```
Hyperparam config #1 of #10: lr: 7.18e-03, reg: 6.30e-07, hidden_size: 82,
epochs: 10
```

```
/content/drive/My Drive/Coursework/ENPM703/assignment1/cs231n/layers.py:240:
RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(p[np.arange(num_train), y])) / num_train
/content/drive/My Drive/Coursework/ENPM703/assignment1/cs231n/layers.py:236:
RuntimeWarning: overflow encountered in exp
  ex = np.exp(x)  # Compute the exponentials of the scores
/content/drive/My Drive/Coursework/ENPM703/assignment1/cs231n/layers.py:237:
RuntimeWarning: invalid value encountered in divide
  p = (ex.T / np.sum(ex, axis=1)).T  # Calculate softmax probabilities
```

```
Validation accuracy: 0.1670

Hyperparam config #2 of #10: lr: 3.73e-03, reg: 2.60e-07, hidden_size: 62,
epochs: 20
Validation accuracy: 0.1800

Hyperparam config #3 of #10: lr: 6.32e-03, reg: 7.76e-04, hidden_size: 78,
epochs: 20
Validation accuracy: 0.2090

Hyperparam config #4 of #10: lr: 4.32e-03, reg: 1.28e-05, hidden_size: 59,
epochs: 20
Validation accuracy: 0.2090

Hyperparam config #5 of #10: lr: 1.91e-03, reg: 1.61e-06, hidden_size: 98,
epochs: 10
Validation accuracy: 0.5020

Hyperparam config #6 of #10: lr: 1.38e-03, reg: 5.47e-07, hidden_size: 78,
epochs: 20
Validation accuracy: 0.5190

Hyperparam config #7 of #10: lr: 6.44e-04, reg: 6.45e-06, hidden_size: 52,
epochs: 10
Validation accuracy: 0.5190

Hyperparam config #8 of #10: lr: 5.42e-03, reg: 3.65e-05, hidden_size: 77,
epochs: 20
Validation accuracy: 0.5190

Hyperparam config #9 of #10: lr: 6.79e-03, reg: 2.51e-04, hidden_size: 50,
epochs: 20
Validation accuracy: 0.5190
```

```
Hyperparam config #10 of #10: lr: 3.18e-04, reg: 4.71e-04, hidden_size: 80,
epochs: 20
Validation accuracy: 0.5370

Best validation accuracy achieved: 0.5370
```

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

```
Validation set accuracy:  0.537
```

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

```
Test set accuracy:  0.524
```

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.

YourAnswer : The correct choices are:

Train on a larger dataset.

Increase the regularization strength.

YourExplanation :

Training on a larger dataset: By exposing the model to a greater variety of examples and lowering overfitting, a larger training dataset improves generalization. For example, in the CIFAR dataset, more samples for each class enable the model to observe a greater range of variances, improving its ability to identify data that was not noticed during testing. But, it's crucial to make sure the extra data isn't noisy because it could interfere with the model's ability to learn.

Increasing the regularization strength: Regularization discourages the model from overfitting by penalizing overly large weights. The model is forced to rely less on "outlier" features or noisy patterns in the training data by making the model less complex. The influence can be distributed more equally across all characteristics with a greater regularization term, which will improve the model's ability to generalize to new data.

Adding more hidden units: While this can seem like a good idea, doing so can complicate the model and cause overfitting. Therefore, in most cases, this is not the ideal way to close the accuracy gap between training and testing.

[ ]:

features

October 6, 2024

[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 = 'Coursework/ENPM703/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/Coursework/ENPM703/assignment1/cs231n/datasets
/content/drive/My Drive/Coursework/ENPM703/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.

[9]:
```
from cs231n.features import *

num_color_bins = 15 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins), lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

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

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

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

```
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
```

```
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

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

[10]:
```
# 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-7]
regularization_strengths = [5e4, 5e5, 5e6]

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)*****

# pass
for lr in learning_rates:
    for rs in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=lr, reg=rs,␣
 ↪num_iters=3000,
                  verbose=False)
        y_train_pred = svm.predict(X_train_feats)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        val_acc = np.mean(y_val == y_val_pred)

        results[lr, rs] = [train_acc, val_acc]
        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm
```

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

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

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

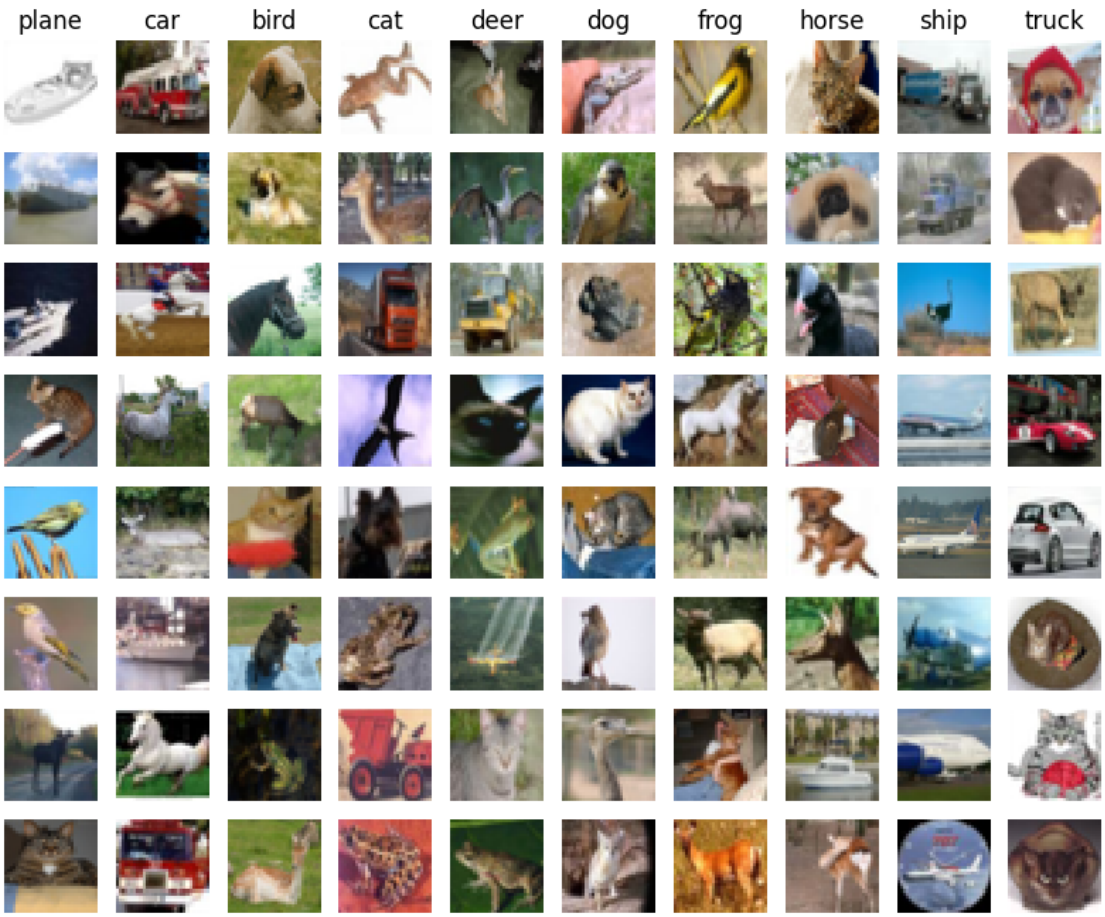
```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.110041 val accuracy: 0.103000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.096918 val accuracy: 0.095000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.422449 val accuracy: 0.426000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.151755 val accuracy: 0.156000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.423592 val accuracy: 0.437000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.418082 val accuracy: 0.414000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.423082 val accuracy: 0.435000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.419735 val accuracy: 0.414000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.340122 val accuracy: 0.343000
best validation accuracy achieved: 0.437000
```

[11]:
```
# 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)
```

```
0.419
```

[12]:
```
# 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?

YourAnswer :

Upon extracting texture and color data, photos with similar forms or colors could be mistakenly labeled. Animal classes, for instance, all have brown coloring, a tail, and four feet. The background colors—white and blue seem to have a greater influence on the type of plane or ship than the shape. However, there are misclassified photos with no evident shared qualities in certain classes, such as frog, truck, and automobile. This could indicate that insufficient features were extracted by the HOG and color histogram methods to properly categorize the pictures.

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.

[13]:
```
# 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, 175)
(49000, 174)
```

[28]:
```
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
}
```

[33]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver
from itertools import product

# Input dimensions and other parameters
input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

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

# Initialize variables to store the best model and its parameters
best_net = None
```

```
best_accuracy = 0.0
best_params = None

# Hyperparameter ranges
learning_rates = [0.1, 0.105]
regularization_strengths = [2e-5, 3e-5]
lr_decay = [0.9, 1.0]

# Create all combinations of hyperparameters
grids = list(product(learning_rates, regularization_strengths, lr_decay))

# Loop through each combination of hyperparameters
total_combinations = len(grids)
for idx, (lr, reg, dec) in enumerate(grids):
    print(f'Starting training with lr={lr}, reg={reg}, decay={dec} (Combination␣
 ↪{idx+1}/{total_combinations})')

    # Create a new instance of the TwoLayerNet for each parameter set
    net = TwoLayerNet(input_dim=input_dim,
                      hidden_dim=hidden_dim,
                      num_classes=num_classes,
                      weight_scale=1e-3,
                      reg=reg)

    # Create a new solver for this combination
    solver = Solver(net, data,
                    update_rule='sgd',
                    optim_config={'learning_rate': lr},
                    lr_decay=dec,
                    num_epochs=14,  # Set the number of epochs to train
                    batch_size=200,
                    print_every=100)

    # Train the model
    solver.train()

    print(f'Finished training with lr={lr}, reg={reg}, decay={dec}\n')

    # Check if this is the best accuracy so far
    final_val_acc = solver.val_acc_history[-1]
    if final_val_acc > best_accuracy:
        best_params = (lr, reg, dec)
        best_net = net
        best_accuracy = final_val_acc

# Print the best results
print(f'Best accuracy: {best_accuracy}')
```

```
print('Best params:', best_params)

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

```
Starting training with lr=0.1, reg=2e-05, decay=0.9 (Combination 1/8)
(Iteration 1 / 3430) loss: 2.302614
(Epoch 0 / 14) train acc: 0.087000; val_acc: 0.107000
(Iteration 101 / 3430) loss: 2.193658
(Iteration 201 / 3430) loss: 1.762734
(Epoch 1 / 14) train acc: 0.445000; val_acc: 0.411000
(Iteration 301 / 3430) loss: 1.467168
(Iteration 401 / 3430) loss: 1.460248
(Epoch 2 / 14) train acc: 0.510000; val_acc: 0.507000
(Iteration 501 / 3430) loss: 1.327508
(Iteration 601 / 3430) loss: 1.295423
(Iteration 701 / 3430) loss: 1.390969
(Epoch 3 / 14) train acc: 0.536000; val_acc: 0.510000
(Iteration 801 / 3430) loss: 1.154440
(Iteration 901 / 3430) loss: 1.282816
(Epoch 4 / 14) train acc: 0.523000; val_acc: 0.514000
(Iteration 1001 / 3430) loss: 1.225342
(Iteration 1101 / 3430) loss: 1.321424
(Iteration 1201 / 3430) loss: 1.305530
(Epoch 5 / 14) train acc: 0.533000; val_acc: 0.525000
(Iteration 1301 / 3430) loss: 1.218100
(Iteration 1401 / 3430) loss: 1.322140
(Epoch 6 / 14) train acc: 0.550000; val_acc: 0.536000
(Iteration 1501 / 3430) loss: 1.248223
(Iteration 1601 / 3430) loss: 1.365719
(Iteration 1701 / 3430) loss: 1.209192
(Epoch 7 / 14) train acc: 0.587000; val_acc: 0.539000
(Iteration 1801 / 3430) loss: 1.289528
(Iteration 1901 / 3430) loss: 1.300189
(Epoch 8 / 14) train acc: 0.571000; val_acc: 0.550000
(Iteration 2001 / 3430) loss: 1.254595
(Iteration 2101 / 3430) loss: 1.283270
(Iteration 2201 / 3430) loss: 1.278447
(Epoch 9 / 14) train acc: 0.567000; val_acc: 0.555000
(Iteration 2301 / 3430) loss: 1.131499
(Iteration 2401 / 3430) loss: 1.195249
(Epoch 10 / 14) train acc: 0.596000; val_acc: 0.548000
(Iteration 2501 / 3430) loss: 1.170244
(Iteration 2601 / 3430) loss: 1.193447
(Epoch 11 / 14) train acc: 0.597000; val_acc: 0.547000
(Iteration 2701 / 3430) loss: 1.153594
(Iteration 2801 / 3430) loss: 1.200873
(Iteration 2901 / 3430) loss: 1.215633
```

```
(Epoch 12 / 14) train acc: 0.616000; val_acc: 0.552000
(Iteration 3001 / 3430) loss: 1.115485
(Iteration 3101 / 3430) loss: 1.147546
(Epoch 13 / 14) train acc: 0.603000; val_acc: 0.563000
(Iteration 3201 / 3430) loss: 1.065170
(Iteration 3301 / 3430) loss: 1.193037
(Iteration 3401 / 3430) loss: 1.218044
(Epoch 14 / 14) train acc: 0.620000; val_acc: 0.570000
Finished training with lr=0.1, reg=2e-05, decay=0.9

Starting training with lr=0.1, reg=2e-05, decay=1.0 (Combination 2/8)
(Iteration 1 / 3430) loss: 2.302597
(Epoch 0 / 14) train acc: 0.105000; val_acc: 0.079000
(Iteration 101 / 3430) loss: 2.227989
(Iteration 201 / 3430) loss: 1.690394
(Epoch 1 / 14) train acc: 0.439000; val_acc: 0.413000
(Iteration 301 / 3430) loss: 1.460359
(Iteration 401 / 3430) loss: 1.310547
(Epoch 2 / 14) train acc: 0.492000; val_acc: 0.510000
(Iteration 501 / 3430) loss: 1.349125
(Iteration 601 / 3430) loss: 1.155509
(Iteration 701 / 3430) loss: 1.286344
(Epoch 3 / 14) train acc: 0.539000; val_acc: 0.525000
(Iteration 801 / 3430) loss: 1.232906
(Iteration 901 / 3430) loss: 1.297864
(Epoch 4 / 14) train acc: 0.550000; val_acc: 0.525000
(Iteration 1001 / 3430) loss: 1.295642
(Iteration 1101 / 3430) loss: 1.220209
(Iteration 1201 / 3430) loss: 1.233288
(Epoch 5 / 14) train acc: 0.529000; val_acc: 0.538000
(Iteration 1301 / 3430) loss: 1.175443
(Iteration 1401 / 3430) loss: 1.321444
(Epoch 6 / 14) train acc: 0.600000; val_acc: 0.546000
(Iteration 1501 / 3430) loss: 1.191472
(Iteration 1601 / 3430) loss: 1.228197
(Iteration 1701 / 3430) loss: 1.079448
(Epoch 7 / 14) train acc: 0.575000; val_acc: 0.540000
(Iteration 1801 / 3430) loss: 1.230929
(Iteration 1901 / 3430) loss: 1.282815
(Epoch 8 / 14) train acc: 0.573000; val_acc: 0.561000
(Iteration 2001 / 3430) loss: 1.156991
(Iteration 2101 / 3430) loss: 1.258586
(Iteration 2201 / 3430) loss: 1.084991
(Epoch 9 / 14) train acc: 0.601000; val_acc: 0.573000
(Iteration 2301 / 3430) loss: 1.087059
(Iteration 2401 / 3430) loss: 1.106512
(Epoch 10 / 14) train acc: 0.625000; val_acc: 0.577000
(Iteration 2501 / 3430) loss: 1.146469
```

```
(Iteration 2601 / 3430) loss: 0.905945
(Epoch 11 / 14) train acc: 0.669000; val_acc: 0.590000
(Iteration 2701 / 3430) loss: 1.049791
(Iteration 2801 / 3430) loss: 0.936341
(Iteration 2901 / 3430) loss: 0.964314
(Epoch 12 / 14) train acc: 0.633000; val_acc: 0.587000
(Iteration 3001 / 3430) loss: 1.095077
(Iteration 3101 / 3430) loss: 0.915235
(Epoch 13 / 14) train acc: 0.659000; val_acc: 0.593000
(Iteration 3201 / 3430) loss: 1.108672
(Iteration 3301 / 3430) loss: 0.883655
(Iteration 3401 / 3430) loss: 1.022680
(Epoch 14 / 14) train acc: 0.667000; val_acc: 0.583000
Finished training with lr=0.1, reg=2e-05, decay=1.0

Starting training with lr=0.1, reg=3e-05, decay=0.9 (Combination 3/8)
(Iteration 1 / 3430) loss: 2.302583
(Epoch 0 / 14) train acc: 0.126000; val_acc: 0.078000
(Iteration 101 / 3430) loss: 2.163743
(Iteration 201 / 3430) loss: 1.689076
(Epoch 1 / 14) train acc: 0.423000; val_acc: 0.409000
(Iteration 301 / 3430) loss: 1.468342
(Iteration 401 / 3430) loss: 1.400123
(Epoch 2 / 14) train acc: 0.502000; val_acc: 0.516000
(Iteration 501 / 3430) loss: 1.344432
(Iteration 601 / 3430) loss: 1.470953
(Iteration 701 / 3430) loss: 1.337389
(Epoch 3 / 14) train acc: 0.511000; val_acc: 0.518000
(Iteration 801 / 3430) loss: 1.378205
(Iteration 901 / 3430) loss: 1.366299
(Epoch 4 / 14) train acc: 0.537000; val_acc: 0.519000
(Iteration 1001 / 3430) loss: 1.479450
(Iteration 1101 / 3430) loss: 1.246330
(Iteration 1201 / 3430) loss: 1.217656
(Epoch 5 / 14) train acc: 0.541000; val_acc: 0.527000
(Iteration 1301 / 3430) loss: 1.272328
(Iteration 1401 / 3430) loss: 1.359915
(Epoch 6 / 14) train acc: 0.562000; val_acc: 0.534000
(Iteration 1501 / 3430) loss: 1.315265
(Iteration 1601 / 3430) loss: 1.185550
(Iteration 1701 / 3430) loss: 1.173259
(Epoch 7 / 14) train acc: 0.564000; val_acc: 0.548000
(Iteration 1801 / 3430) loss: 1.190389
(Iteration 1901 / 3430) loss: 1.245224
(Epoch 8 / 14) train acc: 0.543000; val_acc: 0.553000
(Iteration 2001 / 3430) loss: 1.229295
(Iteration 2101 / 3430) loss: 1.282176
(Iteration 2201 / 3430) loss: 1.165120
```

```
(Epoch 9 / 14) train acc: 0.553000; val_acc: 0.550000
(Iteration 2301 / 3430) loss: 1.185585
(Iteration 2401 / 3430) loss: 1.246728
(Epoch 10 / 14) train acc: 0.589000; val_acc: 0.552000
(Iteration 2501 / 3430) loss: 1.224165
(Iteration 2601 / 3430) loss: 1.047224
(Epoch 11 / 14) train acc: 0.567000; val_acc: 0.555000
(Iteration 2701 / 3430) loss: 1.142951
(Iteration 2801 / 3430) loss: 1.273048
(Iteration 2901 / 3430) loss: 1.297862
(Epoch 12 / 14) train acc: 0.587000; val_acc: 0.561000
(Iteration 3001 / 3430) loss: 1.186532
(Iteration 3101 / 3430) loss: 1.150719
(Epoch 13 / 14) train acc: 0.625000; val_acc: 0.559000
(Iteration 3201 / 3430) loss: 1.167744
(Iteration 3301 / 3430) loss: 1.103216
(Iteration 3401 / 3430) loss: 1.168152
(Epoch 14 / 14) train acc: 0.607000; val_acc: 0.572000
Finished training with lr=0.1, reg=3e-05, decay=0.9

Starting training with lr=0.1, reg=3e-05, decay=1.0 (Combination 4/8)
(Iteration 1 / 3430) loss: 2.302608
(Epoch 0 / 14) train acc: 0.104000; val_acc: 0.098000
(Iteration 101 / 3430) loss: 2.188558
(Iteration 201 / 3430) loss: 1.771813
(Epoch 1 / 14) train acc: 0.426000; val_acc: 0.410000
(Iteration 301 / 3430) loss: 1.536815
(Iteration 401 / 3430) loss: 1.363950
(Epoch 2 / 14) train acc: 0.483000; val_acc: 0.499000
(Iteration 501 / 3430) loss: 1.315995
(Iteration 601 / 3430) loss: 1.424839
(Iteration 701 / 3430) loss: 1.372068
(Epoch 3 / 14) train acc: 0.524000; val_acc: 0.525000
(Iteration 801 / 3430) loss: 1.248104
(Iteration 901 / 3430) loss: 1.267578
(Epoch 4 / 14) train acc: 0.535000; val_acc: 0.528000
(Iteration 1001 / 3430) loss: 1.219429
(Iteration 1101 / 3430) loss: 1.255797
(Iteration 1201 / 3430) loss: 1.360073
(Epoch 5 / 14) train acc: 0.543000; val_acc: 0.527000
(Iteration 1301 / 3430) loss: 1.352549
(Iteration 1401 / 3430) loss: 1.151696
(Epoch 6 / 14) train acc: 0.576000; val_acc: 0.543000
(Iteration 1501 / 3430) loss: 1.159651
(Iteration 1601 / 3430) loss: 1.370666
(Iteration 1701 / 3430) loss: 1.202070
(Epoch 7 / 14) train acc: 0.578000; val_acc: 0.548000
(Iteration 1801 / 3430) loss: 1.125424
```

```
(Iteration 1901 / 3430) loss: 1.237926
(Epoch 8 / 14) train acc: 0.589000; val_acc: 0.560000
(Iteration 2001 / 3430) loss: 1.014974
(Iteration 2101 / 3430) loss: 1.023513
(Iteration 2201 / 3430) loss: 1.120757
(Epoch 9 / 14) train acc: 0.621000; val_acc: 0.567000
(Iteration 2301 / 3430) loss: 1.067049
(Iteration 2401 / 3430) loss: 1.127198
(Epoch 10 / 14) train acc: 0.613000; val_acc: 0.580000
(Iteration 2501 / 3430) loss: 0.922756
(Iteration 2601 / 3430) loss: 1.020564
(Epoch 11 / 14) train acc: 0.633000; val_acc: 0.577000
(Iteration 2701 / 3430) loss: 1.151263
(Iteration 2801 / 3430) loss: 1.027406
(Iteration 2901 / 3430) loss: 1.030689
(Epoch 12 / 14) train acc: 0.645000; val_acc: 0.582000
(Iteration 3001 / 3430) loss: 0.966772
(Iteration 3101 / 3430) loss: 1.001674
(Epoch 13 / 14) train acc: 0.627000; val_acc: 0.595000
(Iteration 3201 / 3430) loss: 0.914820
(Iteration 3301 / 3430) loss: 0.978411
(Iteration 3401 / 3430) loss: 1.068982
(Epoch 14 / 14) train acc: 0.631000; val_acc: 0.586000
Finished training with lr=0.1, reg=3e-05, decay=1.0

Starting training with lr=0.105, reg=2e-05, decay=0.9 (Combination 5/8)
(Iteration 1 / 3430) loss: 2.302603
(Epoch 0 / 14) train acc: 0.102000; val_acc: 0.107000
(Iteration 101 / 3430) loss: 2.171723
(Iteration 201 / 3430) loss: 1.674779
(Epoch 1 / 14) train acc: 0.441000; val_acc: 0.415000
(Iteration 301 / 3430) loss: 1.613394
(Iteration 401 / 3430) loss: 1.512972
(Epoch 2 / 14) train acc: 0.494000; val_acc: 0.525000
(Iteration 501 / 3430) loss: 1.458966
(Iteration 601 / 3430) loss: 1.371676
(Iteration 701 / 3430) loss: 1.337702
(Epoch 3 / 14) train acc: 0.476000; val_acc: 0.511000
(Iteration 801 / 3430) loss: 1.271590
(Iteration 901 / 3430) loss: 1.236564
(Epoch 4 / 14) train acc: 0.535000; val_acc: 0.513000
(Iteration 1001 / 3430) loss: 1.370143
(Iteration 1101 / 3430) loss: 1.237350
(Iteration 1201 / 3430) loss: 1.401596
(Epoch 5 / 14) train acc: 0.552000; val_acc: 0.533000
(Iteration 1301 / 3430) loss: 1.264080
(Iteration 1401 / 3430) loss: 1.205256
(Epoch 6 / 14) train acc: 0.579000; val_acc: 0.532000
```

```
(Iteration 1501 / 3430) loss: 1.131956
(Iteration 1601 / 3430) loss: 1.280149
(Iteration 1701 / 3430) loss: 1.373126
(Epoch 7 / 14) train acc: 0.554000; val_acc: 0.553000
(Iteration 1801 / 3430) loss: 1.201978
(Iteration 1901 / 3430) loss: 1.232882
(Epoch 8 / 14) train acc: 0.567000; val_acc: 0.557000
(Iteration 2001 / 3430) loss: 1.151519
(Iteration 2101 / 3430) loss: 1.226475
(Iteration 2201 / 3430) loss: 1.177356
(Epoch 9 / 14) train acc: 0.565000; val_acc: 0.552000
(Iteration 2301 / 3430) loss: 1.178439
(Iteration 2401 / 3430) loss: 1.063700
(Epoch 10 / 14) train acc: 0.601000; val_acc: 0.562000
(Iteration 2501 / 3430) loss: 1.116368
(Iteration 2601 / 3430) loss: 1.279436
(Epoch 11 / 14) train acc: 0.573000; val_acc: 0.566000
(Iteration 2701 / 3430) loss: 1.063437
(Iteration 2801 / 3430) loss: 1.253792
(Iteration 2901 / 3430) loss: 1.195152
(Epoch 12 / 14) train acc: 0.595000; val_acc: 0.572000
(Iteration 3001 / 3430) loss: 1.128170
(Iteration 3101 / 3430) loss: 1.178364
(Epoch 13 / 14) train acc: 0.595000; val_acc: 0.565000
(Iteration 3201 / 3430) loss: 1.191495
(Iteration 3301 / 3430) loss: 1.221355
(Iteration 3401 / 3430) loss: 1.094962
(Epoch 14 / 14) train acc: 0.609000; val_acc: 0.559000
Finished training with lr=0.105, reg=2e-05, decay=0.9

Starting training with lr=0.105, reg=2e-05, decay=1.0 (Combination 6/8)
(Iteration 1 / 3430) loss: 2.302560
(Epoch 0 / 14) train acc: 0.098000; val_acc: 0.078000
(Iteration 101 / 3430) loss: 2.190908
(Iteration 201 / 3430) loss: 1.693931
(Epoch 1 / 14) train acc: 0.437000; val_acc: 0.422000
(Iteration 301 / 3430) loss: 1.437127
(Iteration 401 / 3430) loss: 1.421262
(Epoch 2 / 14) train acc: 0.529000; val_acc: 0.507000
(Iteration 501 / 3430) loss: 1.385599
(Iteration 601 / 3430) loss: 1.370495
(Iteration 701 / 3430) loss: 1.270331
(Epoch 3 / 14) train acc: 0.556000; val_acc: 0.524000
(Iteration 801 / 3430) loss: 1.439429
(Iteration 901 / 3430) loss: 1.379430
(Epoch 4 / 14) train acc: 0.528000; val_acc: 0.531000
(Iteration 1001 / 3430) loss: 1.275438
(Iteration 1101 / 3430) loss: 1.202138
```

```
(Iteration 1201 / 3430) loss: 1.206743
(Epoch 5 / 14) train acc: 0.565000; val_acc: 0.530000
(Iteration 1301 / 3430) loss: 1.284520
(Iteration 1401 / 3430) loss: 1.320420
(Epoch 6 / 14) train acc: 0.577000; val_acc: 0.546000
(Iteration 1501 / 3430) loss: 1.192121
(Iteration 1601 / 3430) loss: 1.139202
(Iteration 1701 / 3430) loss: 1.172275
(Epoch 7 / 14) train acc: 0.580000; val_acc: 0.554000
(Iteration 1801 / 3430) loss: 1.268427
(Iteration 1901 / 3430) loss: 1.180171
(Epoch 8 / 14) train acc: 0.602000; val_acc: 0.573000
(Iteration 2001 / 3430) loss: 1.091952
(Iteration 2101 / 3430) loss: 1.189447
(Iteration 2201 / 3430) loss: 1.206676
(Epoch 9 / 14) train acc: 0.590000; val_acc: 0.581000
(Iteration 2301 / 3430) loss: 1.132939
(Iteration 2401 / 3430) loss: 1.285435
(Epoch 10 / 14) train acc: 0.621000; val_acc: 0.581000
(Iteration 2501 / 3430) loss: 1.088017
(Iteration 2601 / 3430) loss: 1.106071
(Epoch 11 / 14) train acc: 0.668000; val_acc: 0.591000
(Iteration 2701 / 3430) loss: 0.969874
(Iteration 2801 / 3430) loss: 1.004488
(Iteration 2901 / 3430) loss: 1.041565
(Epoch 12 / 14) train acc: 0.676000; val_acc: 0.596000
(Iteration 3001 / 3430) loss: 0.875766
(Iteration 3101 / 3430) loss: 1.046978
(Epoch 13 / 14) train acc: 0.660000; val_acc: 0.596000
(Iteration 3201 / 3430) loss: 1.002583
(Iteration 3301 / 3430) loss: 0.964623
(Iteration 3401 / 3430) loss: 1.097320
(Epoch 14 / 14) train acc: 0.671000; val_acc: 0.600000
Finished training with lr=0.105, reg=2e-05, decay=1.0

Starting training with lr=0.105, reg=3e-05, decay=0.9 (Combination 7/8)
(Iteration 1 / 3430) loss: 2.302557
(Epoch 0 / 14) train acc: 0.091000; val_acc: 0.113000
(Iteration 101 / 3430) loss: 2.162731
(Iteration 201 / 3430) loss: 1.651970
(Epoch 1 / 14) train acc: 0.418000; val_acc: 0.413000
(Iteration 301 / 3430) loss: 1.581335
(Iteration 401 / 3430) loss: 1.391306
(Epoch 2 / 14) train acc: 0.489000; val_acc: 0.511000
(Iteration 501 / 3430) loss: 1.349664
(Iteration 601 / 3430) loss: 1.376976
(Iteration 701 / 3430) loss: 1.236082
(Epoch 3 / 14) train acc: 0.538000; val_acc: 0.519000
```

```
(Iteration 801 / 3430) loss: 1.305710
(Iteration 901 / 3430) loss: 1.328257
(Epoch 4 / 14) train acc: 0.545000; val_acc: 0.522000
(Iteration 1001 / 3430) loss: 1.353276
(Iteration 1101 / 3430) loss: 1.330127
(Iteration 1201 / 3430) loss: 1.355264
(Epoch 5 / 14) train acc: 0.536000; val_acc: 0.533000
(Iteration 1301 / 3430) loss: 1.218449
(Iteration 1401 / 3430) loss: 1.205231
(Epoch 6 / 14) train acc: 0.567000; val_acc: 0.535000
(Iteration 1501 / 3430) loss: 1.295514
(Iteration 1601 / 3430) loss: 1.202775
(Iteration 1701 / 3430) loss: 1.154774
(Epoch 7 / 14) train acc: 0.552000; val_acc: 0.540000
(Iteration 1801 / 3430) loss: 1.170781
(Iteration 1901 / 3430) loss: 1.317685
(Epoch 8 / 14) train acc: 0.577000; val_acc: 0.554000
(Iteration 2001 / 3430) loss: 1.132028
(Iteration 2101 / 3430) loss: 1.356139
(Iteration 2201 / 3430) loss: 1.153140
(Epoch 9 / 14) train acc: 0.609000; val_acc: 0.560000
(Iteration 2301 / 3430) loss: 1.104897
(Iteration 2401 / 3430) loss: 1.068680
(Epoch 10 / 14) train acc: 0.584000; val_acc: 0.555000
(Iteration 2501 / 3430) loss: 1.249324
(Iteration 2601 / 3430) loss: 1.245545
(Epoch 11 / 14) train acc: 0.591000; val_acc: 0.563000
(Iteration 2701 / 3430) loss: 1.060595
(Iteration 2801 / 3430) loss: 1.190740
(Iteration 2901 / 3430) loss: 1.294188
(Epoch 12 / 14) train acc: 0.604000; val_acc: 0.565000
(Iteration 3001 / 3430) loss: 1.147638
(Iteration 3101 / 3430) loss: 1.098826
(Epoch 13 / 14) train acc: 0.584000; val_acc: 0.569000
(Iteration 3201 / 3430) loss: 1.084883
(Iteration 3301 / 3430) loss: 1.027540
(Iteration 3401 / 3430) loss: 1.192022
(Epoch 14 / 14) train acc: 0.584000; val_acc: 0.573000
Finished training with lr=0.105, reg=3e-05, decay=0.9

Starting training with lr=0.105, reg=3e-05, decay=1.0 (Combination 8/8)
(Iteration 1 / 3430) loss: 2.302566
(Epoch 0 / 14) train acc: 0.110000; val_acc: 0.134000
(Iteration 101 / 3430) loss: 2.192956
(Iteration 201 / 3430) loss: 1.666592
(Epoch 1 / 14) train acc: 0.420000; val_acc: 0.416000
(Iteration 301 / 3430) loss: 1.570090
(Iteration 401 / 3430) loss: 1.467553
```

```
(Epoch 2 / 14) train acc: 0.491000; val_acc: 0.506000
(Iteration 501 / 3430) loss: 1.251476
(Iteration 601 / 3430) loss: 1.406082
(Iteration 701 / 3430) loss: 1.386408
(Epoch 3 / 14) train acc: 0.504000; val_acc: 0.512000
(Iteration 801 / 3430) loss: 1.342895
(Iteration 901 / 3430) loss: 1.368153
(Epoch 4 / 14) train acc: 0.533000; val_acc: 0.519000
(Iteration 1001 / 3430) loss: 1.313851
(Iteration 1101 / 3430) loss: 1.347048
(Iteration 1201 / 3430) loss: 1.371877
(Epoch 5 / 14) train acc: 0.548000; val_acc: 0.547000
(Iteration 1301 / 3430) loss: 1.170076
(Iteration 1401 / 3430) loss: 1.194125
(Epoch 6 / 14) train acc: 0.551000; val_acc: 0.549000
(Iteration 1501 / 3430) loss: 1.268427
(Iteration 1601 / 3430) loss: 1.333999
(Iteration 1701 / 3430) loss: 1.094171
(Epoch 7 / 14) train acc: 0.600000; val_acc: 0.560000
(Iteration 1801 / 3430) loss: 1.156506
(Iteration 1901 / 3430) loss: 1.289420
(Epoch 8 / 14) train acc: 0.588000; val_acc: 0.564000
(Iteration 2001 / 3430) loss: 1.184438
(Iteration 2101 / 3430) loss: 1.009227
(Iteration 2201 / 3430) loss: 1.018827
(Epoch 9 / 14) train acc: 0.626000; val_acc: 0.559000
(Iteration 2301 / 3430) loss: 1.169506
(Iteration 2401 / 3430) loss: 1.055063
(Epoch 10 / 14) train acc: 0.642000; val_acc: 0.579000
(Iteration 2501 / 3430) loss: 1.018684
(Iteration 2601 / 3430) loss: 1.004853
(Epoch 11 / 14) train acc: 0.629000; val_acc: 0.584000
(Iteration 2701 / 3430) loss: 0.935330
(Iteration 2801 / 3430) loss: 0.957688
(Iteration 2901 / 3430) loss: 1.086421
(Epoch 12 / 14) train acc: 0.611000; val_acc: 0.588000
(Iteration 3001 / 3430) loss: 1.113842
(Iteration 3101 / 3430) loss: 0.918672
(Epoch 13 / 14) train acc: 0.675000; val_acc: 0.591000
(Iteration 3201 / 3430) loss: 0.941731
(Iteration 3301 / 3430) loss: 1.000715
(Iteration 3401 / 3430) loss: 0.973093
(Epoch 14 / 14) train acc: 0.694000; val_acc: 0.598000
Finished training with lr=0.105, reg=3e-05, decay=1.0

Best accuracy: 0.6
Best params: (0.105, 2e-05, 1.0)
```

[34]:
```
# 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(solver.model.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.565
```