knn

April 22, 2024

[ ]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/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.

[ ]:

```
# Run some setup code for this notebook.

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

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

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

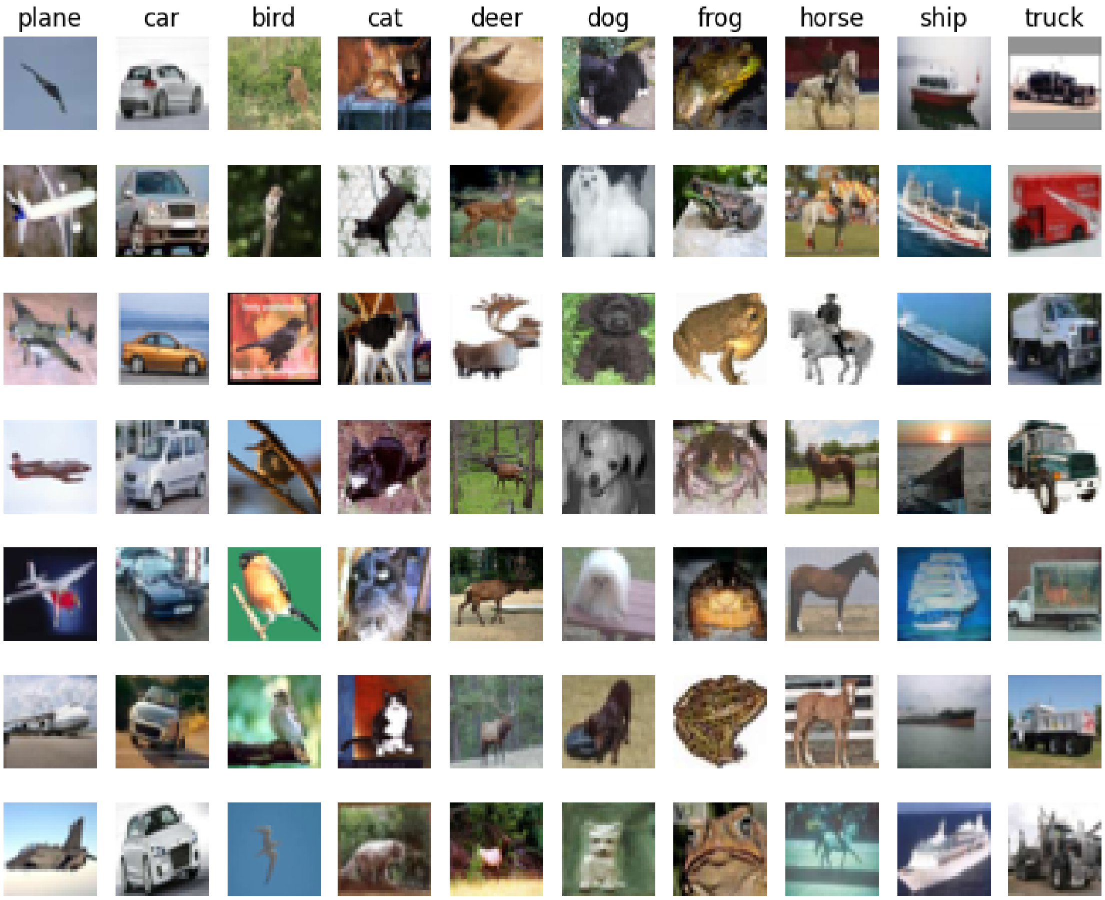
[ ]:

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

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

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

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

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

[ ]:

```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()
```



[ ]:

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

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

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

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

[ ]:

```
from cs231n.classifiers import KNearestNeighbor

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

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

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

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

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

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

[ ]:

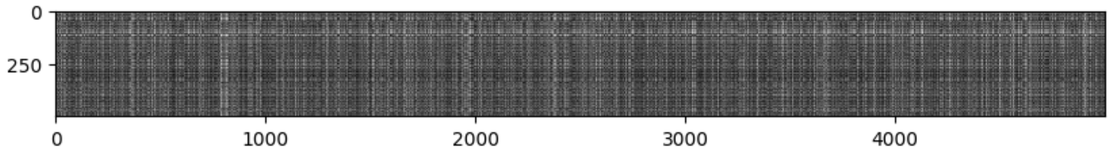
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
print(dists.shape)
```

```
(500, 5000)
```

[ ]:

```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

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

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

YourAnswer :

Distinctly Bright Rows: As white indicates high distances, the distinctly bright rows possibly indicate test examples are significantly different from all training examples. It indicate this is an outlier in the test set, either a novel class which can hardly be represented in the training set, or image with large noises.

Distinctly Bright Columns: Similarly, bright columns might indicate training examples are very different from all test examples, suggeesting an outlier or noises.

[ ]:

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

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

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

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

[ ]:

```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
```

```
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. To clarify, both training and test examples are preprocessed in the same way.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

$Your Answer :$ 1, 2, 3

$Your Explanation :$

1. Subtracting the mean μ: This operation merely shifts all data points by a globally same value and won't change the relative distances between any pair of points. Hence, it doesn't affec the performance of a k-NN classifier using L1 distance.

2. Subtracting the per pixel mean μ_{ij}: This does not change the relative distances between data points for L1 distance. Though each pixel has a different mean, the relative distance between each correponding pixel won't change.

3. Subtracting the mean μ and dividing by the standard deviation σ: Since the substraction and scaling are uniform, and the relative L1 distances between points stay the same. Therefore, the performance of the k-NN classifier does not change.

4. **Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}** This process normalize the data accross the pixel, meaning the contribution of diffent pixel to the total distance is normalized. Thus, the performance is affected.

5. Rotating the coordinate axes of the data: This process can lead to new pixel values in the empty regions and we are lossing pixel information in some regions due to the rotation, thus change the value of final L1 distance

[ ]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

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

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

[ ]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

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

[ ]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
```

```
Two loop version took 31.504951 seconds
One loop version took 47.047667 seconds
No loop version took 0.544146 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.

[ ]:

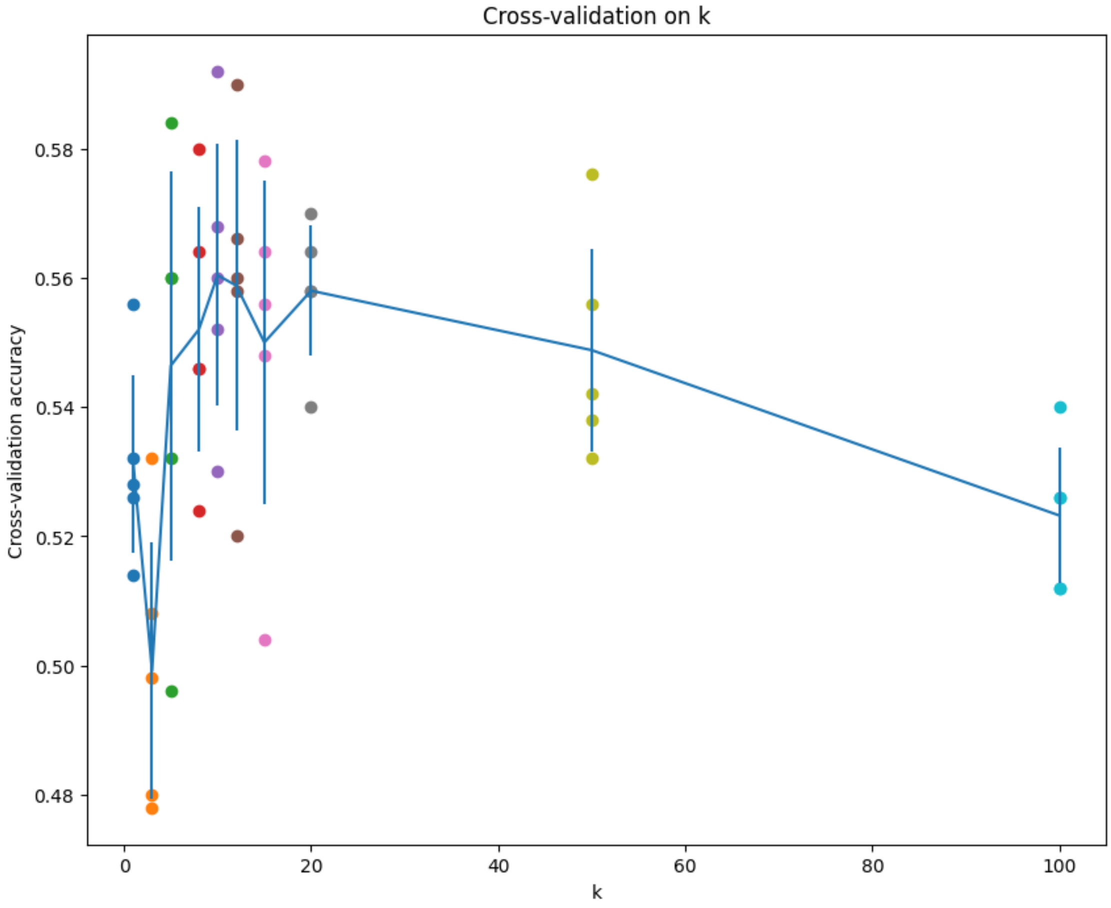
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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

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

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

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


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

for k_choice in k_choices:
  k_to_accuracies[k_choice] = []
  for fold in range(num_folds):
    X_train_new = np.concatenate(X_train_folds[:fold] + X_train_folds[fold + 1:
 ↪])
    y_train_new = np.concatenate(y_train_folds[:fold] + y_train_folds[fold + 1:
 ↪])
    classifier = KNearestNeighbor()
    classifier.train(X_train_new, y_train_new)
    dists = classifier.compute_distances_no_loops(X_train_folds[fold])
    y_test_pred = classifier.predict_labels(dists, k = k_choice)
    num_correct = np.sum(y_test_pred == y_train_folds[fold])
    k_to_accuracies[k_choice].append(float(num_correct) / num_test)


# *****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))
```

```
(5000, 3072)
k = 1, accuracy = 0.526000
k = 1, accuracy = 0.514000
k = 1, accuracy = 0.528000
k = 1, accuracy = 0.556000
k = 1, accuracy = 0.532000
k = 3, accuracy = 0.478000
k = 3, accuracy = 0.498000
k = 3, accuracy = 0.480000
k = 3, accuracy = 0.532000
k = 3, accuracy = 0.508000
k = 5, accuracy = 0.496000
k = 5, accuracy = 0.532000
k = 5, accuracy = 0.560000
k = 5, accuracy = 0.584000
k = 5, accuracy = 0.560000
k = 8, accuracy = 0.524000
k = 8, accuracy = 0.564000
k = 8, accuracy = 0.546000
k = 8, accuracy = 0.580000
k = 8, accuracy = 0.546000
k = 10, accuracy = 0.530000
k = 10, accuracy = 0.592000
k = 10, accuracy = 0.552000
k = 10, accuracy = 0.568000
k = 10, accuracy = 0.560000
k = 12, accuracy = 0.520000
k = 12, accuracy = 0.590000
k = 12, accuracy = 0.558000
k = 12, accuracy = 0.566000
k = 12, accuracy = 0.560000
k = 15, accuracy = 0.504000
k = 15, accuracy = 0.578000
k = 15, accuracy = 0.556000
k = 15, accuracy = 0.564000
k = 15, accuracy = 0.548000
k = 20, accuracy = 0.540000
k = 20, accuracy = 0.558000
k = 20, accuracy = 0.558000
k = 20, accuracy = 0.564000
k = 20, accuracy = 0.570000
k = 50, accuracy = 0.542000
k = 50, accuracy = 0.576000
k = 50, accuracy = 0.556000
k = 50, accuracy = 0.538000
k = 50, accuracy = 0.532000
k = 100, accuracy = 0.512000
k = 100, accuracy = 0.540000
```

```
k = 100, accuracy = 0.526000
k = 100, accuracy = 0.512000
k = 100, accuracy = 0.526000
```

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



[ ]:

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

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

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

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than 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. No, k-NN can result in non-linear boundaries because the distribution of the data is not guaranteed to be linear.
2. The training error of a 1-NN can be lower than that of 5-NN, but it's not guaranteed to be lower or equal for all datasets since 1-NN can overfit the training data.
3. No, 1-NN might have higher test error due to overfitting, and a larger k value might fitting the data better.
4. The training time should be linear to the size of training set because we are computing the distance from the test example to each example in the training set.

svm

April 22, 2024

[ ]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

- **visualize** the final learned weights

[ ]:

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

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

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

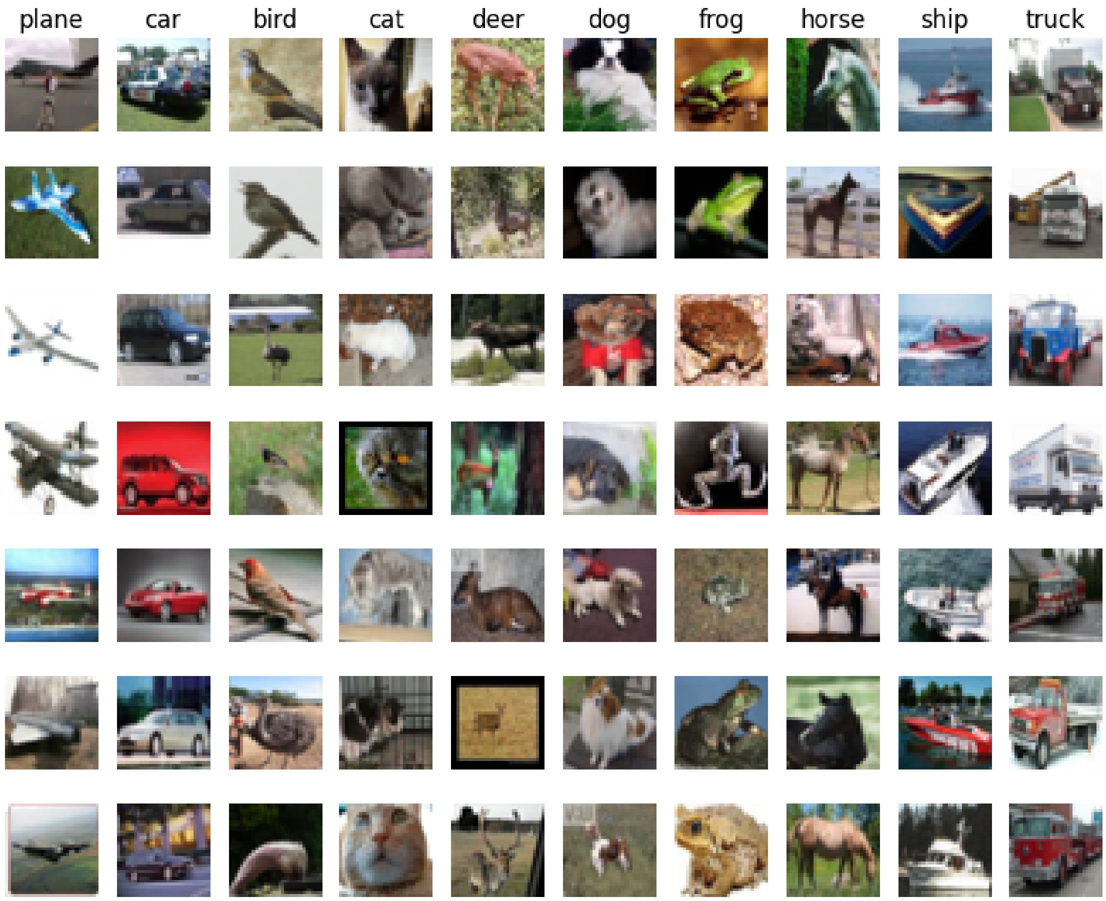
[ ]:

```
# 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)
```

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

[ ]:

```
# 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,)
```

[ ]:

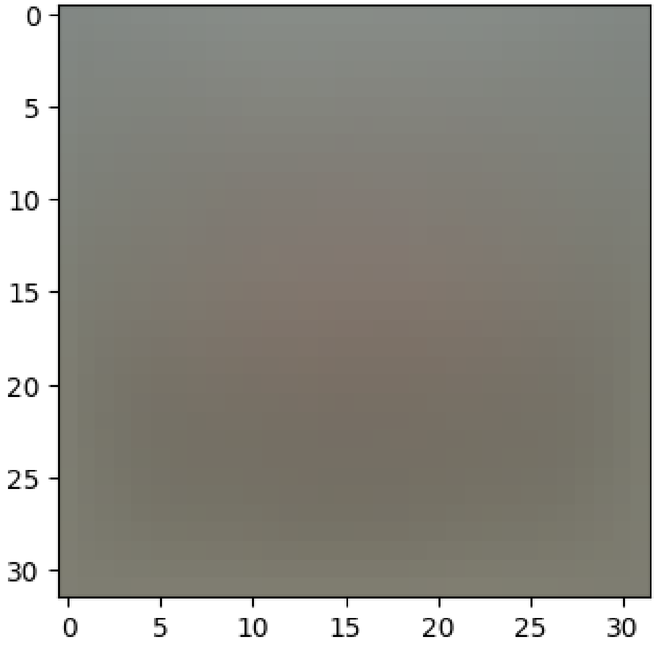
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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: 9.167102
```

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, 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:

[ ]:

```python
# 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: -53.546676 analytic: -53.546676, relative error: 3.134146e-12
numerical: 17.050853 analytic: 17.050853, relative error: 1.023568e-12
numerical: -0.797138 analytic: -0.797138, relative error: 2.842031e-10
numerical: -4.325579 analytic: -4.325579, relative error: 4.604963e-11
numerical: 11.339243 analytic: 11.339243, relative error: 4.881164e-11
numerical: -11.675037 analytic: -11.675037, relative error: 5.104176e-12
numerical: -6.054780 analytic: -6.054780, relative error: 1.709655e-11
numerical: 7.334116 analytic: 7.334116, relative error: 5.286419e-12
numerical: 8.511548 analytic: 8.511548, relative error: 8.137616e-12
numerical: -0.248442 analytic: -0.248442, relative error: 1.578351e-09
numerical: 18.037888 analytic: 18.037888, relative error: 1.066106e-11
numerical: -11.529760 analytic: -11.529760, relative error: 8.020888e-12
numerical: 7.084150 analytic: 7.051919, relative error: 2.280086e-03
numerical: -14.347205 analytic: -14.363043, relative error: 5.516256e-04
numerical: 0.111002 analytic: 0.111002, relative error: 5.865995e-10
numerical: -3.156519 analytic: -3.156519, relative error: 7.458515e-12
numerical: -35.290982 analytic: -35.245113, relative error: 6.503032e-04
numerical: 31.457927 analytic: 31.457927, relative error: 1.076873e-11
numerical: 15.757488 analytic: 15.757488, relative error: 3.008626e-11
numerical: -11.472890 analytic: -11.472890, relative error: 1.144958e-11
```

Inline Question 1

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

YourAnswer : Yes, it is possible. The reason is that for the SVM loss function, when the score for a class is exactly on the margin(around 1), since this point is non-differentiable, so the analytical gradient may not match the numerical gradient.

This typically won't be a cause for concern, as the relative error are small at most time. As mentioned above, it fails when the point is very close to 1.

Increasing margin cause more examples to end up being close or exactly on the margin, thus increasing the frequency of encountering such discrepency. On the other hand, decreasing margin leave more buffer space, and reduce the frequency of encountering such discrepency.

[ ]:

```python
# 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: 9.167102e+00 computed in 0.153227s
Vectorized loss: 9.167102e+00 computed in 0.004586s
difference: 0.000000
```

[ ]:

```python
# 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 0.304051s
Vectorized loss and gradient: computed in 0.018641s
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`.

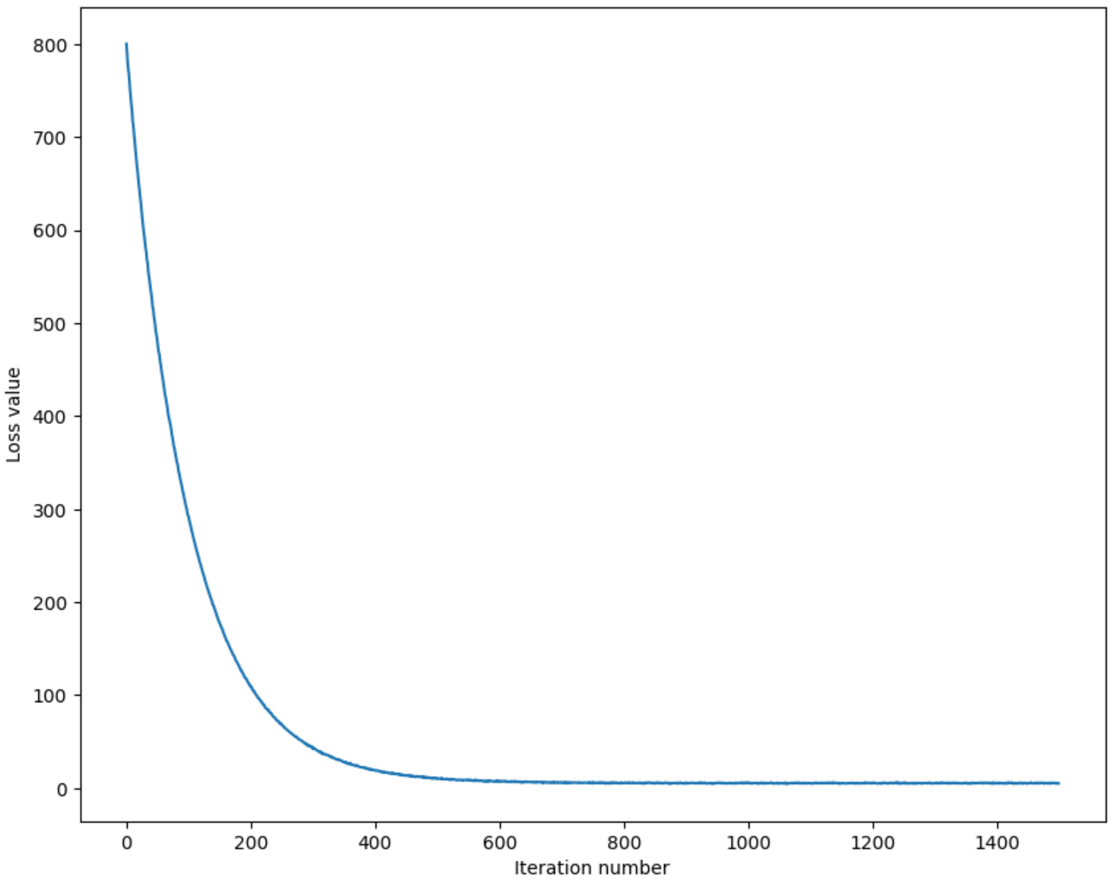
[ ]:

```
# 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 800.419815
iteration 100 / 1500: loss 292.027967
iteration 200 / 1500: loss 108.958419
iteration 300 / 1500: loss 43.796540
iteration 400 / 1500: loss 18.976788
iteration 500 / 1500: loss 10.741158
iteration 600 / 1500: loss 7.012888
iteration 700 / 1500: loss 5.698306
iteration 800 / 1500: loss 5.394865
iteration 900 / 1500: loss 6.074024
iteration 1000 / 1500: loss 5.466225
iteration 1100 / 1500: loss 5.045919
iteration 1200 / 1500: loss 5.194876
iteration 1300 / 1500: loss 5.345012
iteration 1400 / 1500: loss 5.426311
That took 15.297626s
```

[ ]:

```
# 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: 0.366510
validation accuracy: 0.379000
```

[ ]:

```
# 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 (> 0.385) 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
```

```python
# (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]
regularization_strengths = [2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
  for regu_strength in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate=lr, reg=regu_strength,
                      num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)
    train_accuracy, val_accuracy = np.mean(y_train == y_train_pred), np.mean(y_val == y_val_pred)
    results[(lr, regu_strength)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_svm = svm
      best_val = val_accuracy


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

# Print out results.
```

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

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

```
iteration 0 / 1500: loss 793.211411
iteration 100 / 1500: loss 289.629503
iteration 200 / 1500: loss 108.907874
iteration 300 / 1500: loss 42.639056
iteration 400 / 1500: loss 18.982336
iteration 500 / 1500: loss 10.003001
iteration 600 / 1500: loss 6.787335
iteration 700 / 1500: loss 5.927627
iteration 800 / 1500: loss 5.616007
iteration 900 / 1500: loss 5.869276
iteration 1000 / 1500: loss 4.610142
iteration 1100 / 1500: loss 5.158060
iteration 1200 / 1500: loss 4.760345
iteration 1300 / 1500: loss 4.354906
iteration 1400 / 1500: loss 5.011995
iteration 0 / 1500: loss 1566.189811
iteration 100 / 1500: loss 211.687104
iteration 200 / 1500: loss 33.091870
iteration 300 / 1500: loss 9.433162
iteration 400 / 1500: loss 6.352074
iteration 500 / 1500: loss 6.031348
iteration 600 / 1500: loss 5.205931
iteration 700 / 1500: loss 6.097299
iteration 800 / 1500: loss 5.723452
iteration 900 / 1500: loss 5.609563
iteration 1000 / 1500: loss 6.020252
iteration 1100 / 1500: loss 5.571482
iteration 1200 / 1500: loss 5.972986
iteration 1300 / 1500: loss 5.712839
iteration 1400 / 1500: loss 6.018354
iteration 0 / 1500: loss 787.433730
iteration 100 / 1500: loss 438431354710961180402370580134731186176.000000
iteration 200 / 1500: loss 7246919157156067598783046042549484207596193486088752242176839728333062144 0.000000
iteration 300 / 1500: loss 119785769667359538302661542564233571422732755511067356630959173333871159621509917575992993585069707988052213 76.000000
iteration 400 / 1500: loss 197996283712271887245720378833800344760241550086227669368230369021813563622006340395645147039722592617679542399124558822054701094824439 2684552192.000000
iteration 500 / 1500: loss 32727199961009049031432813368876914330701952120080859
23203174555920569630143650283891059095407014897288411969349400395347934301873082
37996568464953586801972382318506231999011749888.000000
iteration 600 / 1500: loss 54095440439899779673995760407915114158778660643413485
13416386470047037380850455793206148879058837673105765828779868069550391537100791
70714432824725140001384516498519854903948185327474796841814920812059785471941672
96.000000
iteration 700 / 1500: loss 89415430585969400038189856481012684655501249640269541
96391449373816839753259563147985194779409146125364221750679230060659262261985452
00544942287570573092774729396086479143941076908599025116756190705672957397722429
3330922487335426033426134291758186496.000000
iteration 800 / 1500: loss 14779654554725214068755028457486769732752004929776139
91926288196346740071045759818275809738743780296157551179366229678876088884937031
69327453442155721909560617825182344550812730002165833989250688805445345894787271
5556617374788313774860062665950253475565717890903398956513485339374911488.000000
```

```
/content/drive/MyDrive/cs231n/assignments/assignment1/cs231n/classifiers/linear_
svm.py:89: RuntimeWarning: overflow encountered in scalar multiply
  loss = np.sum(L_i)/num_train + 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/MyDrive/cs231n/assignments/assignment1/cs231n/classifiers/linear_
svm.py:89: RuntimeWarning: overflow encountered in multiply
  loss = np.sum(L_i)/num_train + reg * np.sum(W * W)
```

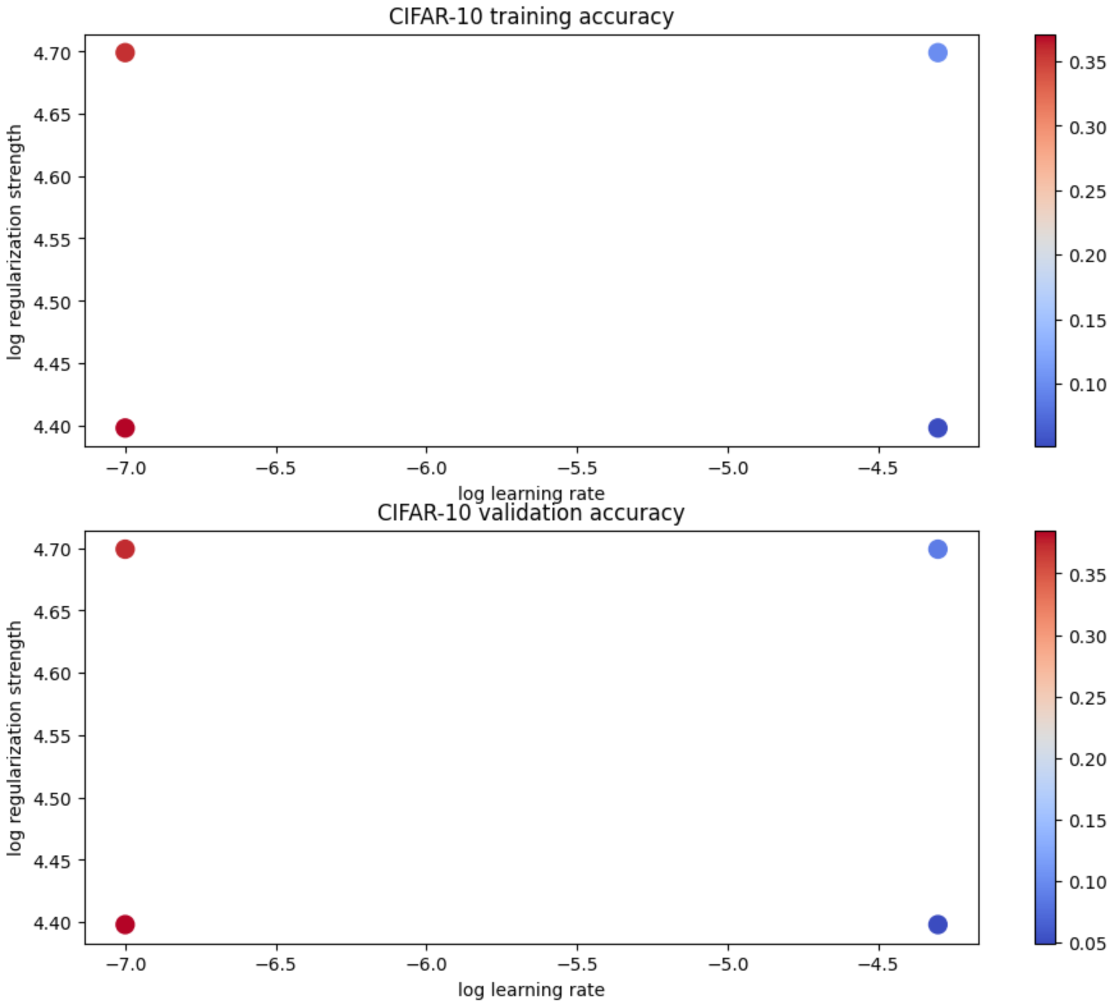
```
iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
iteration 0 / 1500: loss 1569.386865
iteration 100 / 1500: loss 42754945961989331822982704039465963496956472120070539
4194940204567330436037218693580209637699803778186271117504678349871513 6.000000
iteration 200 / 1500: loss 11040395399795931679157013431823087453101783052085942
98873409949100617192366761997853670388937305492100060161431601327388783662606249
23931608889446428977639363970135934265995995251468520806120107398046550925498571
8185804064041600276386067093913 6.000000
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf
```

```
/content/drive/MyDrive/cs231n/assignments/assignment1/cs231n/classifiers/linear_
svm.py:110: RuntimeWarning: overflow encountered in multiply
  dW = dW/num_train + 2 * reg * W
/content/drive/MyDrive/cs231n/assignments/assignment1/cs231n/classifiers/linear_
svm.py:84: RuntimeWarning: invalid value encountered in matmul
  scores = X @ W
```

```
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.371143 val accuracy: 0.385000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.357041 val accuracy: 0.372000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.050939 val accuracy: 0.049000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.385000
```

[ ]:

```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()
```

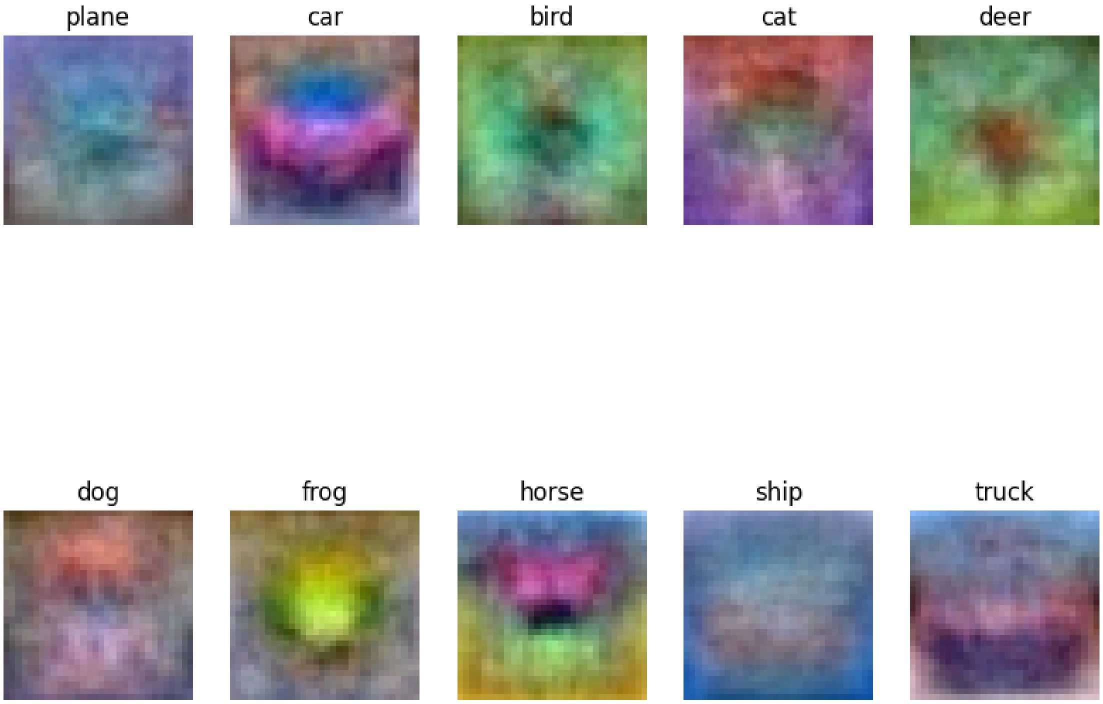


[ ]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.366000
```

[ ]:
```
# 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 the way they do.

YourAnswer : As we can see, though these visualizations looks blurry and abstract compared to a normal image probably because SVM tend to extract the most distinctive features of a class, including the edges, colors, background and so forth. They are blurry since it doesn't represent a single object rather a general class pattern. For instance, for the 'plane' class, the SVM appeare to recognize the typical shape of a plane in the background of blue sky.

softmax

April 22, 2024

[2]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Softmax exercise

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

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

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

[3]:

```
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
```

[4]:

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

```python
    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`.

[5]:

```
# 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: 2.367877
sanity check: 2.302585
```

Inline Question 1

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

$YourAnswer:$ Since we are calculating using the formula $-\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$, and all weight is initially equal, and we are only passing the training data one epoch without updating the weight yet. The change of loss at this stage only rises from the addition of regularization term. So, we are expecting the result after this single training to be close to -log(1/10) = -log(0.1), where 10 is the number of classes in our case

[6]:

```
# 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: 2.557651 analytic: 2.557651, relative error: 8.880312e-09
numerical: 0.159566 analytic: 0.159566, relative error: 1.726899e-07
numerical: -0.993613 analytic: -0.993613, relative error: 9.391684e-09
```

```
numerical: 1.476207 analytic: 1.476207, relative error: 1.335293e-08
numerical: 0.788343 analytic: 0.788343, relative error: 2.331169e-08
numerical: -0.417993 analytic: -0.417993, relative error: 1.026006e-07
numerical: 0.978437 analytic: 0.978437, relative error: 4.904006e-08
numerical: 2.151883 analytic: 2.151883, relative error: 2.927982e-08
numerical: 0.223287 analytic: 0.223287, relative error: 5.729545e-08
numerical: -0.052596 analytic: -0.052597, relative error: 1.329452e-06
numerical: -0.512827 analytic: -0.512827, relative error: 9.453645e-08
numerical: 3.296977 analytic: 3.296977, relative error: 4.648878e-09
numerical: -1.991294 analytic: -1.991294, relative error: 4.001525e-09
numerical: 0.786060 analytic: 0.786060, relative error: 6.238454e-08
numerical: 1.028497 analytic: 1.028497, relative error: 1.804447e-08
numerical: -1.859086 analytic: -1.859086, relative error: 2.942020e-09
numerical: 2.312579 analytic: 2.312579, relative error: 1.031820e-08
numerical: 2.902355 analytic: 2.902355, relative error: 1.452167e-09
numerical: 0.464357 analytic: 0.464357, relative error: 1.342604e-08
numerical: 1.884724 analytic: 1.884724, relative error: 1.277669e-08
```

[7]:

```
# 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: 2.367877e+00 computed in 0.098589s
vectorized loss: 2.367877e+00 computed in 0.017637s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[13]:

```
# 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, 7e-7]
regularization_strengths = [2e4, 5e4]

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

for lr in learning_rates:
  for regu_strength in regularization_strengths:
    softmax = Softmax()
    softmax.train(X_train, y_train, learning_rate=lr, reg=regu_strength,
                          num_iters=1500, verbose=True)
    y_train_pred = softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)
    train_accuracy, val_accuracy = np.mean(y_train == y_train_pred), np.
 ↪mean(y_val == y_val_pred)
    results[(lr, regu_strength)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_softmax = softmax
      best_val = val_accuracy

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

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

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

```
iteration 0 / 1500: loss 620.358370
iteration 100 / 1500: loss 278.119828
iteration 200 / 1500: loss 125.493016
iteration 300 / 1500: loss 57.241567
iteration 400 / 1500: loss 26.786048
iteration 500 / 1500: loss 13.123381
iteration 600 / 1500: loss 6.972388
iteration 700 / 1500: loss 4.260953
iteration 800 / 1500: loss 3.060853
iteration 900 / 1500: loss 2.466854
iteration 1000 / 1500: loss 2.286052
iteration 1100 / 1500: loss 2.187256
iteration 1200 / 1500: loss 2.104071
iteration 1300 / 1500: loss 2.103790
iteration 1400 / 1500: loss 2.044540
iteration 0 / 1500: loss 1550.643284
iteration 100 / 1500: loss 208.781245
iteration 200 / 1500: loss 29.717130
iteration 300 / 1500: loss 5.823955
iteration 400 / 1500: loss 2.686578
iteration 500 / 1500: loss 2.219411
iteration 600 / 1500: loss 2.178932
iteration 700 / 1500: loss 2.160658
iteration 800 / 1500: loss 2.158374
iteration 900 / 1500: loss 2.173095
iteration 1000 / 1500: loss 2.157582
iteration 1100 / 1500: loss 2.165417
iteration 1200 / 1500: loss 2.130668
iteration 1300 / 1500: loss 2.128998
iteration 1400 / 1500: loss 2.177772
iteration 0 / 1500: loss 615.255654
iteration 100 / 1500: loss 4.119343
iteration 200 / 1500: loss 2.072464
iteration 300 / 1500: loss 2.074290
iteration 400 / 1500: loss 2.095534
iteration 500 / 1500: loss 2.129372
iteration 600 / 1500: loss 2.063802
iteration 700 / 1500: loss 2.095147
iteration 800 / 1500: loss 1.954999
iteration 900 / 1500: loss 2.096615
iteration 1000 / 1500: loss 2.064498
iteration 1100 / 1500: loss 2.054988
iteration 1200 / 1500: loss 1.996953
iteration 1300 / 1500: loss 2.065401
iteration 1400 / 1500: loss 2.101611
iteration 0 / 1500: loss 1554.240244
iteration 100 / 1500: loss 2.180944
iteration 200 / 1500: loss 2.133491
iteration 300 / 1500: loss 2.163912
iteration 400 / 1500: loss 2.166259
iteration 500 / 1500: loss 2.134450
iteration 600 / 1500: loss 2.141692
iteration 700 / 1500: loss 2.118173
iteration 800 / 1500: loss 2.161426
iteration 900 / 1500: loss 2.101319
iteration 1000 / 1500: loss 2.176635
iteration 1100 / 1500: loss 2.180675
iteration 1200 / 1500: loss 2.131847
iteration 1300 / 1500: loss 2.189243
iteration 1400 / 1500: loss 2.141456
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.337245 val accuracy: 0.351000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.311735 val accuracy: 0.325000
lr 7.000000e-07 reg 2.000000e+04 train accuracy: 0.328653 val accuracy: 0.355000
lr 7.000000e-07 reg 5.000000e+04 train accuracy: 0.302571 val accuracy: 0.320000
best validation accuracy achieved during cross-validation: 0.355000
```

[14]:

```python
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.342000
```

Inline Question 2 - *True or False*

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

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

$\color{blue}{\textit{Your Explanation :}}$

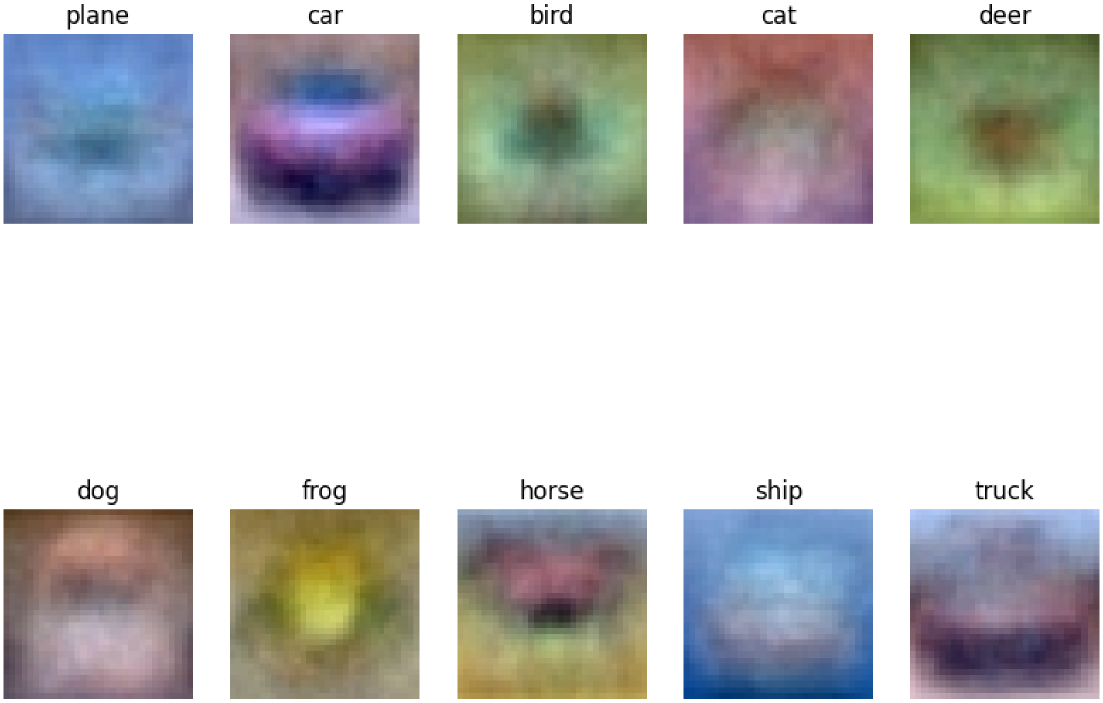
For SVM, it's possible to add a new data point that doesn't change the loss as long as the point is correctly classified and outside the margin as the total loss contribution is zero.

For Softmax classifier, since the softmax function computes the normalized exponential scores for all classes, any new data point will affect the loss. In other word, any new datapoint, no matter it is classified correcetly or not will change the probability distribution and thus change the final loss.

[15]:

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

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



[ ]:

```
```

two_layer_net

April 22, 2024

[1]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Fully-Connected Neural Nets

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

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

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

    return out, cache
```

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

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

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

    return dx, dw
```

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

[2]:

```
# 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))))
```

[3]:
```
# 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:

[4]:
```
# 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.

[5]:

```
# 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:

[6]:

```
# 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:

[7]:

```
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:

Both Sigmoid and ReLU have the vanishing gradident problem. For sigmoid this occurs when input goes to minus or plus infinity, as its derivative $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$ becomes zero in both cases. For ReLU, this occurs when the input is non-positive, where derivative is zero.

We won't have this issue for leaky ReLU, as it is designed to avoid the vanising gradient problem by introducing small positive gradient when the unit is not in the active range.

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:

[8]:
```
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:

[9]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
```

```
y = np.random.randint(num_classes, size=num_inputs)

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

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

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

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

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

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

8 Two-layer network

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

[10]:

```
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)
```

```python
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. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

[14]:

```
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)*****

solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-4,
                    },
                    lr_decay=0.95,
                    num_epochs=5, batch_size=200,
                    print_every=100)
solver.train()

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

```
(Iteration 1 / 1225) loss: 2.302208
(Epoch 0 / 5) train acc: 0.089000; val_acc: 0.103000
(Iteration 101 / 1225) loss: 2.269872
```

```
(Iteration 201 / 1225) loss: 2.180729
(Epoch 1 / 5) train acc: 0.225000; val_acc: 0.243000
(Iteration 301 / 1225) loss: 2.132823
(Iteration 401 / 1225) loss: 2.040491
(Epoch 2 / 5) train acc: 0.302000; val_acc: 0.303000
(Iteration 501 / 1225) loss: 2.002180
(Iteration 601 / 1225) loss: 1.968356
(Iteration 701 / 1225) loss: 1.817955
(Epoch 3 / 5) train acc: 0.322000; val_acc: 0.324000
(Iteration 801 / 1225) loss: 1.951235
(Iteration 901 / 1225) loss: 1.856623
(Epoch 4 / 5) train acc: 0.351000; val_acc: 0.357000
(Iteration 1001 / 1225) loss: 1.741587
(Iteration 1101 / 1225) loss: 1.835495
(Iteration 1201 / 1225) loss: 1.846962
(Epoch 5 / 5) train acc: 0.356000; val_acc: 0.370000
```

10 Debug the training

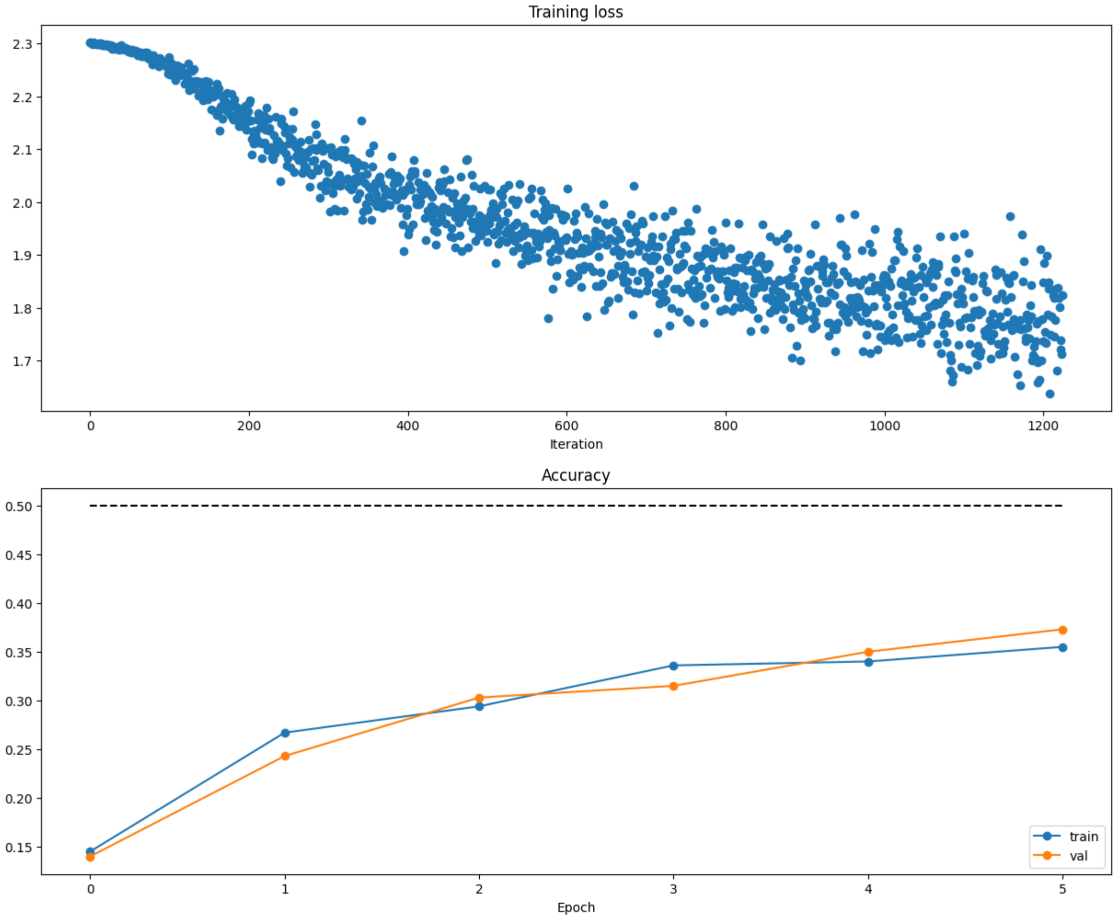
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

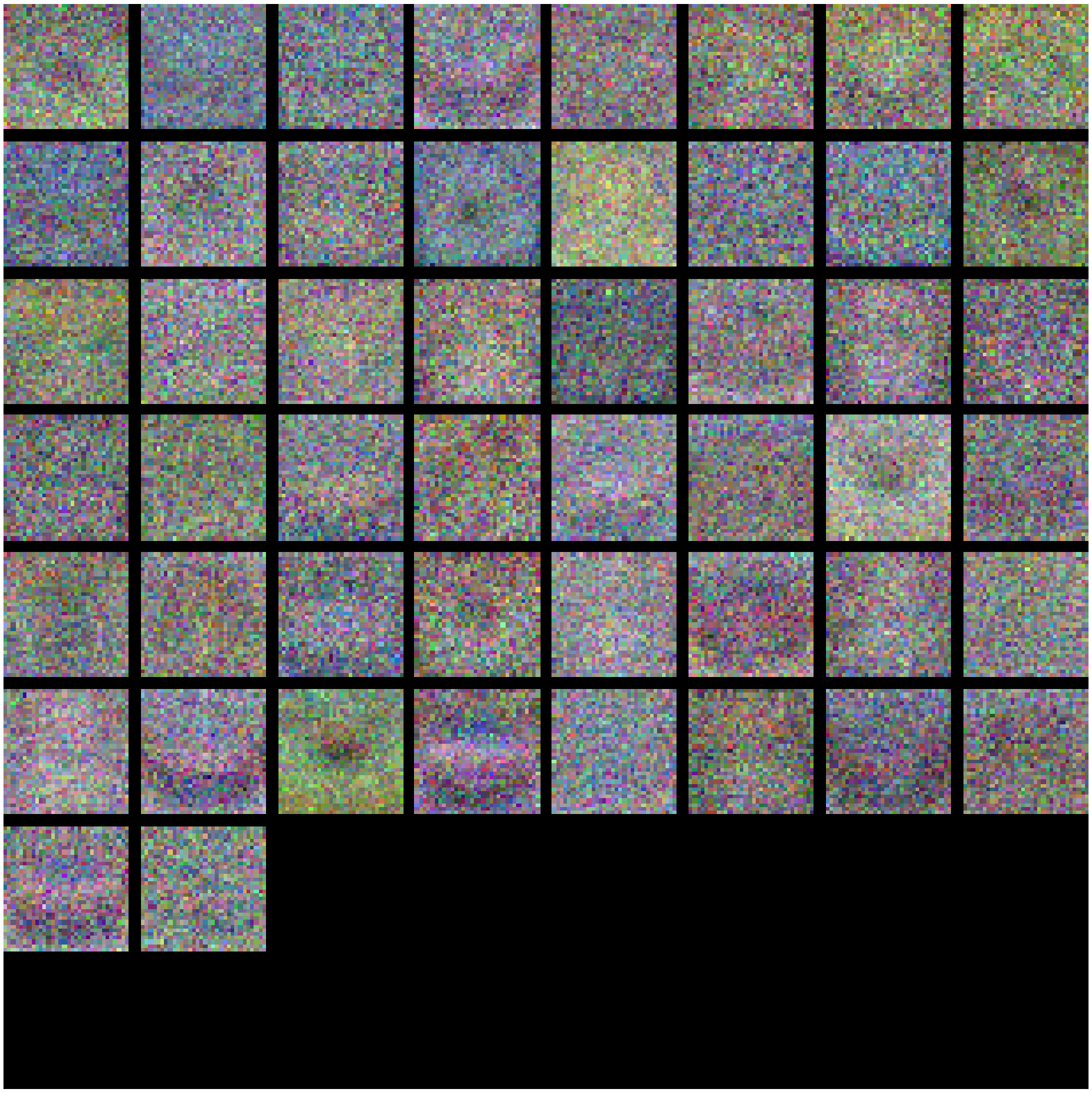
[12]:

```python
# 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()
```



[13]:

```
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(3, 32, 32, -1).transpose(3, 1, 2, 0)
    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.).

[28]:

```
best_model = None


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

learning_rates = [2.5e-4, 5e-4]
hidden_layer_sizes = [80, 120]
input_size = 32 * 32 * 3
num_classes = 10

results = {}
best_val = -1

for lr in learning_rates:
  for layer_size in hidden_layer_sizes:
    model = TwoLayerNet(input_size, layer_size, num_classes)
    solver = Solver(model, data,
                        update_rule='sgd',
                        optim_config={
                          'learning_rate': lr,
                        },
                        lr_decay=0.95,
                        num_epochs=5, batch_size=200,
                        print_every=100)
    solver.train()
    train_accuracy = solver.train_acc_history[-1]
    val_accuracy = solver.val_acc_history[-1]
    results[(lr, layer_size)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_model = model
      best_val = val_accuracy

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)

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

```
(Iteration 1 / 1225) loss: 2.303488
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.103000
(Iteration 101 / 1225) loss: 2.124608
(Iteration 201 / 1225) loss: 1.975125
(Epoch 1 / 5) train acc: 0.335000; val_acc: 0.331000
(Iteration 301 / 1225) loss: 1.833111
(Iteration 401 / 1225) loss: 1.837216
(Epoch 2 / 5) train acc: 0.395000; val_acc: 0.390000
(Iteration 501 / 1225) loss: 1.706329
(Iteration 601 / 1225) loss: 1.785746
(Iteration 701 / 1225) loss: 1.570783
(Epoch 3 / 5) train acc: 0.433000; val_acc: 0.406000
(Iteration 801 / 1225) loss: 1.733729
(Iteration 901 / 1225) loss: 1.709307
(Epoch 4 / 5) train acc: 0.440000; val_acc: 0.441000
(Iteration 1001 / 1225) loss: 1.639428
(Iteration 1101 / 1225) loss: 1.607287
```

```
(Iteration 1201 / 1225) loss: 1.512247
(Epoch 5 / 5) train acc: 0.442000; val_acc: 0.455000
(Iteration 1 / 1225) loss: 2.300319
(Epoch 0 / 5) train acc: 0.098000; val_acc: 0.131000
(Iteration 101 / 1225) loss: 2.099071
(Iteration 201 / 1225) loss: 1.845380
(Epoch 1 / 5) train acc: 0.339000; val_acc: 0.332000
(Iteration 301 / 1225) loss: 1.748747
(Iteration 401 / 1225) loss: 1.833990
(Epoch 2 / 5) train acc: 0.393000; val_acc: 0.390000
(Iteration 501 / 1225) loss: 1.764621
(Iteration 601 / 1225) loss: 1.686599
(Iteration 701 / 1225) loss: 1.624255
(Epoch 3 / 5) train acc: 0.411000; val_acc: 0.430000
(Iteration 801 / 1225) loss: 1.626036
(Iteration 901 / 1225) loss: 1.655015
(Epoch 4 / 5) train acc: 0.432000; val_acc: 0.458000
(Iteration 1001 / 1225) loss: 1.549129
(Iteration 1101 / 1225) loss: 1.462459
(Iteration 1201 / 1225) loss: 1.513758
(Epoch 5 / 5) train acc: 0.448000; val_acc: 0.466000
(Iteration 1 / 1225) loss: 2.303488
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.106000
(Iteration 101 / 1225) loss: 2.031249
(Iteration 201 / 1225) loss: 1.869541
(Epoch 1 / 5) train acc: 0.374000; val_acc: 0.395000
(Iteration 301 / 1225) loss: 1.697491
(Iteration 401 / 1225) loss: 1.686343
(Epoch 2 / 5) train acc: 0.438000; val_acc: 0.428000
(Iteration 501 / 1225) loss: 1.592989
(Iteration 601 / 1225) loss: 1.598965
(Iteration 701 / 1225) loss: 1.470179
(Epoch 3 / 5) train acc: 0.491000; val_acc: 0.463000
(Iteration 801 / 1225) loss: 1.607396
(Iteration 901 / 1225) loss: 1.543526
(Epoch 4 / 5) train acc: 0.489000; val_acc: 0.463000
(Iteration 1001 / 1225) loss: 1.520697
(Iteration 1101 / 1225) loss: 1.468365
(Iteration 1201 / 1225) loss: 1.378488
(Epoch 5 / 5) train acc: 0.495000; val_acc: 0.473000
(Iteration 1 / 1225) loss: 2.300319
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.143000
(Iteration 101 / 1225) loss: 1.956971
(Iteration 201 / 1225) loss: 1.700798
(Epoch 1 / 5) train acc: 0.384000; val_acc: 0.382000
(Iteration 301 / 1225) loss: 1.594564
(Iteration 401 / 1225) loss: 1.738859
(Epoch 2 / 5) train acc: 0.439000; val_acc: 0.446000
```

```
(Iteration 501 / 1225) loss: 1.593279
(Iteration 601 / 1225) loss: 1.582881
(Iteration 701 / 1225) loss: 1.512284
(Epoch 3 / 5) train acc: 0.474000; val_acc: 0.464000
(Iteration 801 / 1225) loss: 1.504072
(Iteration 901 / 1225) loss: 1.525027
(Epoch 4 / 5) train acc: 0.486000; val_acc: 0.491000
(Iteration 1001 / 1225) loss: 1.394742
(Iteration 1101 / 1225) loss: 1.292613
(Iteration 1201 / 1225) loss: 1.371904
(Epoch 5 / 5) train acc: 0.512000; val_acc: 0.487000
lr 2.500000e-04 reg 8.000000e+01 train accuracy: 0.442000 val accuracy: 0.455000
lr 2.500000e-04 reg 1.200000e+02 train accuracy: 0.448000 val accuracy: 0.466000
lr 5.000000e-04 reg 8.000000e+01 train accuracy: 0.495000 val accuracy: 0.473000
lr 5.000000e-04 reg 1.200000e+02 train accuracy: 0.512000 val accuracy: 0.487000
best validation accuracy achieved during cross-validation: 0.487000
```

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.

[29]:
```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.491
```

[30]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.46
```

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 : 1, 3

YourExplanation :

Both training on a larger dataset and increasing the regularization strength can help the model generalize better and reduce overfit during training stage and thus decreasing gap.

While adding more hidden units mightactually intensify overfitting as we are introducing more parameters to fit the data, and even increasing the gap in the end.

[ ]:

```
```

features

April 22, 2024

[1]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Image features exercise

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

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

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

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


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

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

1.1 Load data

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

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

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

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

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

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

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

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

1.2 Extract Features

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

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

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

[4]:

```
from cs231n.features import *

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

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

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

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

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

[7]:

```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-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)*****

for lr in learning_rates:
  for regu_strength in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train_feats, y_train, learning_rate=lr, reg=regu_strength,
                          num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train_feats)
    y_val_pred = svm.predict(X_val_feats)
    train_accuracy, val_accuracy = np.mean(y_train == y_train_pred), np.
  ↪mean(y_val == y_val_pred)
    results[(lr, regu_strength)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_svm = svm
      best_val = val_accuracy

# *****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)
```

```
iteration 0 / 1500: loss 81.152029
iteration 100 / 1500: loss 79.737724
iteration 200 / 1500: loss 78.336552
iteration 300 / 1500: loss 76.971048
iteration 400 / 1500: loss 75.615281
iteration 500 / 1500: loss 74.292532
iteration 600 / 1500: loss 73.008778
iteration 700 / 1500: loss 71.723790
iteration 800 / 1500: loss 70.495769
iteration 900 / 1500: loss 69.273424
iteration 1000 / 1500: loss 68.085628
iteration 1100 / 1500: loss 66.912939
iteration 1200 / 1500: loss 65.764399
iteration 1300 / 1500: loss 64.648229
iteration 1400 / 1500: loss 63.559345
iteration 0 / 1500: loss 754.942026
iteration 100 / 1500: loss 619.675044
iteration 200 / 1500: loss 508.922451
iteration 300 / 1500: loss 418.256248
iteration 400 / 1500: loss 344.038884
iteration 500 / 1500: loss 283.277607
iteration 600 / 1500: loss 233.532760
iteration 700 / 1500: loss 192.816398
iteration 800 / 1500: loss 159.483161
iteration 900 / 1500: loss 132.192489
iteration 1000 / 1500: loss 109.855050
iteration 1100 / 1500: loss 91.558806
iteration 1200 / 1500: loss 76.588920
iteration 1300 / 1500: loss 64.332057
iteration 1400 / 1500: loss 54.298387
iteration 0 / 1500: loss 7753.795039
iteration 100 / 1500: loss 1046.643503
iteration 200 / 1500: loss 148.024149
iteration 300 / 1500: loss 27.626432
iteration 400 / 1500: loss 11.495742
iteration 500 / 1500: loss 9.334318
iteration 600 / 1500: loss 9.044805
iteration 700 / 1500: loss 9.005993
iteration 800 / 1500: loss 9.000799
iteration 900 / 1500: loss 9.000104
iteration 1000 / 1500: loss 9.000010
iteration 1100 / 1500: loss 8.999999
iteration 1200 / 1500: loss 8.999997
iteration 1300 / 1500: loss 8.999996
iteration 1400 / 1500: loss 8.999997
iteration 0 / 1500: loss 87.604892
iteration 100 / 1500: loss 73.366471
iteration 200 / 1500: loss 61.674133
iteration 300 / 1500: loss 52.139521
iteration 400 / 1500: loss 44.314978
iteration 500 / 1500: loss 37.905408
iteration 600 / 1500: loss 32.662887
iteration 700 / 1500: loss 28.377025
iteration 800 / 1500: loss 24.858233
iteration 900 / 1500: loss 21.982845
iteration 1000 / 1500: loss 19.627430
iteration 1100 / 1500: loss 17.704549
iteration 1200 / 1500: loss 16.126047
iteration 1300 / 1500: loss 14.834324
iteration 1400 / 1500: loss 13.774012
iteration 0 / 1500: loss 780.326822
iteration 100 / 1500: loss 112.340343
iteration 200 / 1500: loss 22.847117
iteration 300 / 1500: loss 10.854633
iteration 400 / 1500: loss 9.248255
iteration 500 / 1500: loss 9.033346
iteration 600 / 1500: loss 9.004408
iteration 700 / 1500: loss 9.000554
iteration 800 / 1500: loss 9.000047
iteration 900 / 1500: loss 8.999978
iteration 1000 / 1500: loss 8.999967
iteration 1100 / 1500: loss 8.999959
iteration 1200 / 1500: loss 8.999960
iteration 1300 / 1500: loss 8.999967
iteration 1400 / 1500: loss 8.999962
iteration 0 / 1500: loss 8267.378274
iteration 100 / 1500: loss 9.000002
iteration 200 / 1500: loss 8.999997
iteration 300 / 1500: loss 8.999996
iteration 400 / 1500: loss 8.999996
iteration 500 / 1500: loss 8.999997
iteration 600 / 1500: loss 8.999996
iteration 700 / 1500: loss 8.999997
iteration 800 / 1500: loss 8.999996
iteration 900 / 1500: loss 8.999998
iteration 1000 / 1500: loss 8.999996
iteration 1100 / 1500: loss 8.999996
iteration 1200 / 1500: loss 8.999997
iteration 1300 / 1500: loss 8.999996
iteration 1400 / 1500: loss 8.999996
iteration 0 / 1500: loss 82.243721
iteration 100 / 1500: loss 18.816822
iteration 200 / 1500: loss 10.314527
iteration 300 / 1500: loss 9.175813
iteration 400 / 1500: loss 9.023283
iteration 500 / 1500: loss 9.002807
iteration 600 / 1500: loss 9.000082
iteration 700 / 1500: loss 8.999704
iteration 800 / 1500: loss 8.999583
iteration 900 / 1500: loss 8.999657
iteration 1000 / 1500: loss 8.999626
iteration 1100 / 1500: loss 8.999627
iteration 1200 / 1500: loss 8.999670
iteration 1300 / 1500: loss 8.999630
iteration 1400 / 1500: loss 8.999591
iteration 0 / 1500: loss 769.240013
iteration 100 / 1500: loss 8.999964
iteration 200 / 1500: loss 8.999964
iteration 300 / 1500: loss 8.999973
iteration 400 / 1500: loss 8.999967
iteration 500 / 1500: loss 8.999974
iteration 600 / 1500: loss 8.999967
iteration 700 / 1500: loss 8.999966
iteration 800 / 1500: loss 8.999971
iteration 900 / 1500: loss 8.999965
iteration 1000 / 1500: loss 8.999969
iteration 1100 / 1500: loss 8.999971
iteration 1200 / 1500: loss 8.999968
iteration 1300 / 1500: loss 8.999958
iteration 1400 / 1500: loss 8.999962
iteration 0 / 1500: loss 7844.954858
iteration 100 / 1500: loss 9.000000
iteration 200 / 1500: loss 9.000000
iteration 300 / 1500: loss 9.000000
iteration 400 / 1500: loss 9.000001
iteration 500 / 1500: loss 9.000000
iteration 600 / 1500: loss 8.999999
iteration 700 / 1500: loss 8.999999
iteration 800 / 1500: loss 9.000000
iteration 900 / 1500: loss 9.000001
iteration 1000 / 1500: loss 9.000001
iteration 1100 / 1500: loss 8.999999
iteration 1200 / 1500: loss 9.000000
iteration 1300 / 1500: loss 9.000000
iteration 1400 / 1500: loss 9.000000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.111327 val accuracy: 0.101000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.094755 val accuracy: 0.119000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414265 val accuracy: 0.415000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.102612 val accuracy: 0.098000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.414796 val accuracy: 0.416000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.396245 val accuracy: 0.383000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415673 val accuracy: 0.409000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.399286 val accuracy: 0.392000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.331000 val accuracy: 0.343000
best validation accuracy achieved: 0.416000
```

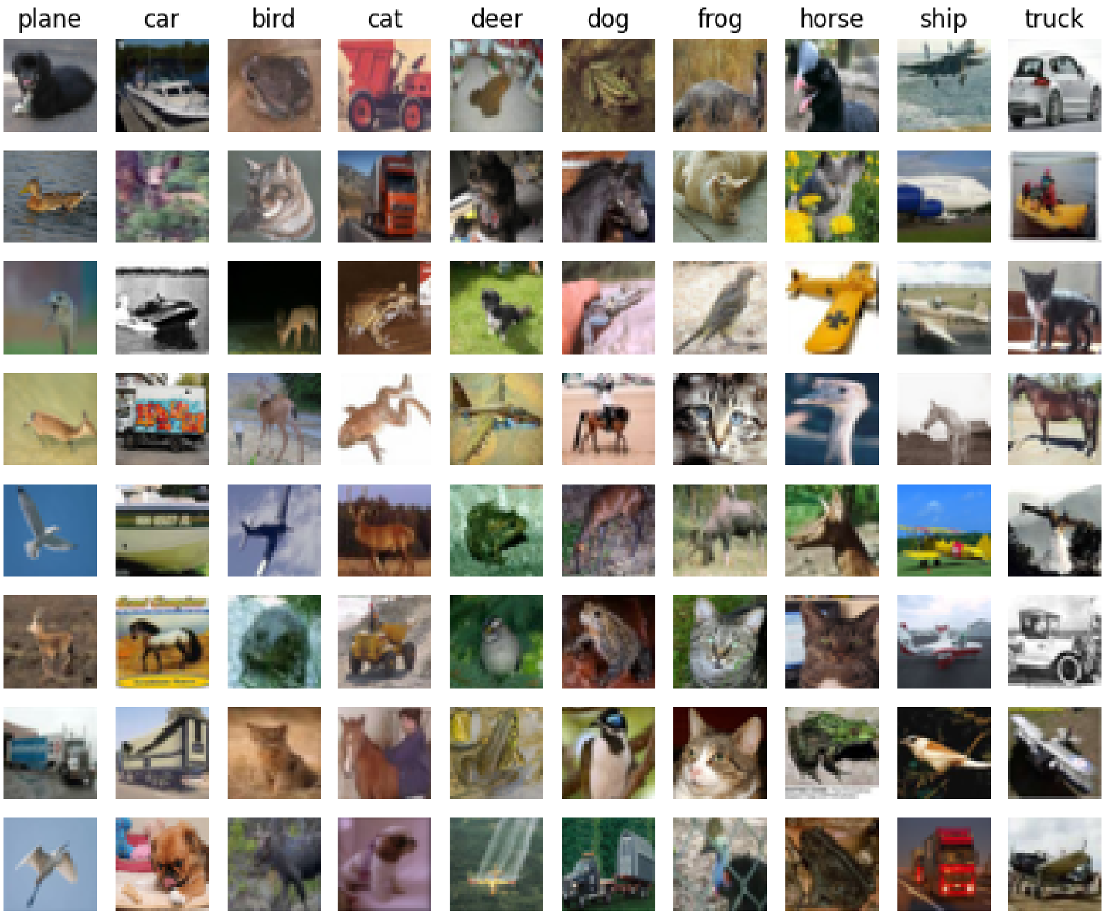
[8]:

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

```
0.418
```

[9]:

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

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



1.3.1 Inline question 1:

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

YourAnswer : We can somehow figure out the possible reason why the misclassification happens. For instance, the alogrithm misclassify bird as plane possibly because both classes share similar environment, color patterns and shapes: blue sky, wing-like feature. Same situation for similar classes including truck and car, frog and cat, etc. Due to this overlapping features, misclassification occurs.

1.4 Neural Network on image features

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

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

[10]:

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

print(X_train_feats.shape)
```

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

[31]:

```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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

data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

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

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

learning_rates = [2e-1, 6e-1]
results = {}
best_val = -1

for lr in learning_rates:
    net = TwoLayerNet(input_dim, hidden_dim, num_classes)
    solver = Solver(net, data,
                        update_rule='sgd',
                        optim_config={
                          'learning_rate': lr,
                        },
                        lr_decay=0.95,
                        num_epochs=8, batch_size=200,
                        print_every=100)
    solver.train()
    train_accuracy = solver.train_acc_history[-1]
    val_accuracy = solver.val_acc_history[-1]
    results[(lr, hidden_dim)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_net = net
      best_val = val_accuracy

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))

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

```
(Iteration 1 / 1960) loss: 2.302586
(Epoch 0 / 8) train acc: 0.103000; val_acc: 0.119000
(Iteration 101 / 1960) loss: 1.828784
(Iteration 201 / 1960) loss: 1.307308
(Epoch 1 / 8) train acc: 0.492000; val_acc: 0.485000
(Iteration 301 / 1960) loss: 1.397196
(Iteration 401 / 1960) loss: 1.403772
(Epoch 2 / 8) train acc: 0.527000; val_acc: 0.509000
(Iteration 501 / 1960) loss: 1.290048
(Iteration 601 / 1960) loss: 1.307009
(Iteration 701 / 1960) loss: 1.221899
(Epoch 3 / 8) train acc: 0.564000; val_acc: 0.557000
(Iteration 801 / 1960) loss: 1.201386
(Iteration 901 / 1960) loss: 1.305643
(Epoch 4 / 8) train acc: 0.579000; val_acc: 0.559000
(Iteration 1001 / 1960) loss: 1.268662
(Iteration 1101 / 1960) loss: 1.084198
(Iteration 1201 / 1960) loss: 1.196222
(Epoch 5 / 8) train acc: 0.606000; val_acc: 0.571000
(Iteration 1301 / 1960) loss: 1.017151
(Iteration 1401 / 1960) loss: 1.136311
(Epoch 6 / 8) train acc: 0.594000; val_acc: 0.571000
(Iteration 1501 / 1960) loss: 1.101106
(Iteration 1601 / 1960) loss: 1.072649
(Iteration 1701 / 1960) loss: 1.006283
(Epoch 7 / 8) train acc: 0.628000; val_acc: 0.593000
(Iteration 1801 / 1960) loss: 0.970249
```

```
(Iteration 1901 / 1960) loss: 1.045015
(Epoch 8 / 8) train acc: 0.611000; val_acc: 0.604000
(Iteration 1 / 1960) loss: 2.302586
(Epoch 0 / 8) train acc: 0.103000; val_acc: 0.119000
(Iteration 101 / 1960) loss: 1.462576
(Iteration 201 / 1960) loss: 1.179175
(Epoch 1 / 8) train acc: 0.492000; val_acc: 0.480000
(Iteration 301 / 1960) loss: 1.261038
(Iteration 401 / 1960) loss: 1.284227
(Epoch 2 / 8) train acc: 0.587000; val_acc: 0.573000
(Iteration 501 / 1960) loss: 1.177770
(Iteration 601 / 1960) loss: 1.137305
(Iteration 701 / 1960) loss: 1.163629
(Epoch 3 / 8) train acc: 0.641000; val_acc: 0.567000
(Iteration 801 / 1960) loss: 1.000069
(Iteration 901 / 1960) loss: 1.218221
(Epoch 4 / 8) train acc: 0.654000; val_acc: 0.590000
(Iteration 1001 / 1960) loss: 1.045150
(Iteration 1101 / 1960) loss: 0.857851
(Iteration 1201 / 1960) loss: 0.912848
(Epoch 5 / 8) train acc: 0.686000; val_acc: 0.576000
(Iteration 1301 / 1960) loss: 0.796591
(Iteration 1401 / 1960) loss: 0.914347
(Epoch 6 / 8) train acc: 0.687000; val_acc: 0.573000
(Iteration 1501 / 1960) loss: 0.811019
(Iteration 1601 / 1960) loss: 0.901911
(Iteration 1701 / 1960) loss: 0.691935
(Epoch 7 / 8) train acc: 0.711000; val_acc: 0.578000
(Iteration 1801 / 1960) loss: 0.693713
(Iteration 1901 / 1960) loss: 0.789811
(Epoch 8 / 8) train acc: 0.726000; val_acc: 0.588000
lr 2.000000e-01 reg 5.000000e+02 train accuracy: 0.611000 val accuracy: 0.604000
lr 6.000000e-01 reg 5.000000e+02 train accuracy: 0.726000 val accuracy: 0.588000
```

[32]:

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

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

```
0.576
```