knn

September 11, 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 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

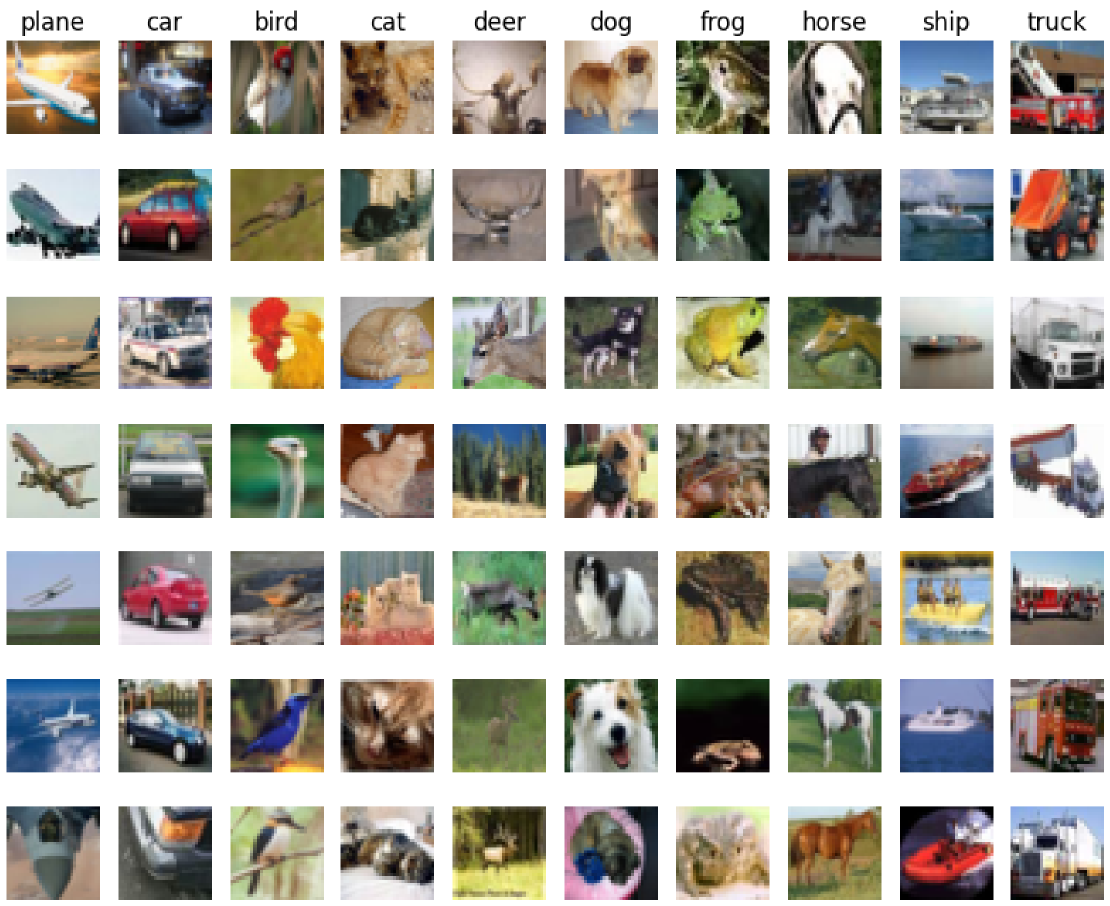
In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

[ ]:
```python
# 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
```

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

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

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

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

```
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()
```



[ ]:
```
# 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.

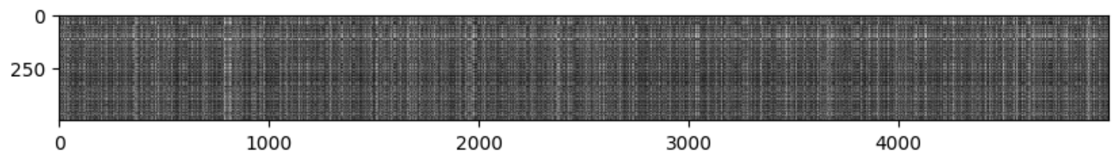
First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

[ ]:
```
# 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?

$Your Answer:$ *fill this in.*

1. The distinctly bright rows indicates that the test data points is far way from most of the training data points. These bright rows could be indicated as outliers in the test set that is totally diferent from the training set. So the euclidean distance for the test points are larger distances to nearly all traning points.

2. Bright columns tells us that the training points are different from the most of the test points. The reason for that could be when the training data contains outliers and its far away from the test points.

[ ]:
```
# 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.

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

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

1. The first process shifts(uniform) all pixel values by the same amount and L1 distance measures absolute differences, so the shift cancels out.So the test and training pints will be shifted by the same constant.Hence not effect the classifier's performance.

2. The second process subtracts a constant value from each pixel position across all images. The L1 distance between any two images at each pixel position remains the same. This will not effect the classifier's performance.

3. The third process changes the scale but does it uniformly acorss all images. For L1 distance, this effictevely multiploes all distances by the same constant , which doesnt change the relative ordering of neighbors not effecting the classifie's performance.

4. When we divide by the standard deviation σ_{ij} for each pixel, we're basically changing the scale of different parts of the image differently. This could mess with which images are considered "neighbors" or similar, and that might change how well our classifier works.

5. Rotating the images changes the spatial arrangement of pixels. This disrupts pixel-to-pixel comparison by rearranging pixels, thus changing the distances between the original and rotated images which will effect the classifier's performance.

[ ]:
```
# 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 38.806811 seconds
One loop version took 39.634400 seconds
No loop version took 0.607682 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.

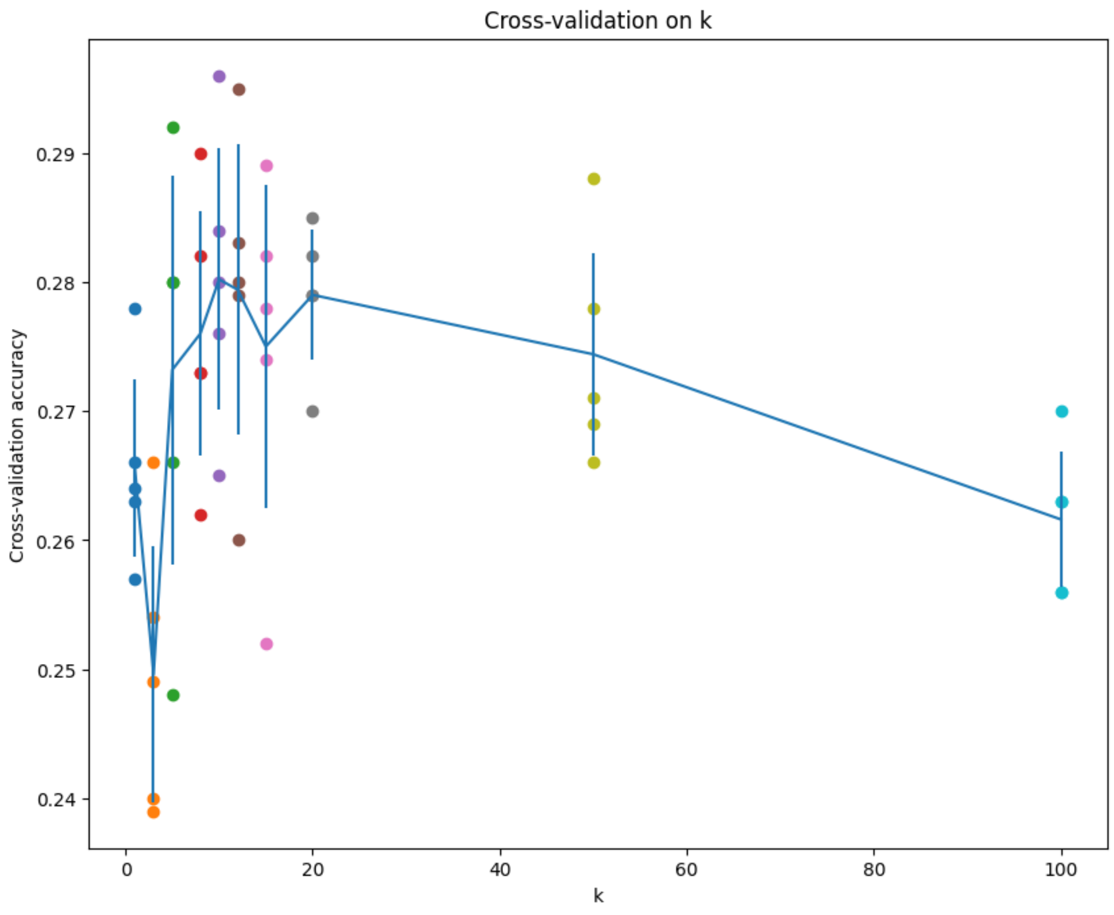
[ ]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)*****
```

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

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

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


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for k in k_choices:
    k_to_accuracies[k] = []
    for fold in range(0, len(y_train_folds)):
        fold_train_X = X_train_folds[:]
        fold_train_y = y_train_folds[:]

        fold_test_X = np.array(fold_train_X.pop(fold))
        fold_test_y = np.array(fold_train_y.pop(fold))
        fold_train_X = np.vstack(fold_train_X)
        fold_train_y = np.hstack(fold_train_y)

        classifier = KNearestNeighbor()
        classifier.train(fold_train_X, fold_train_y)

        fold_test_y_pred = classifier.predict(fold_test_X, k=k)
        accuracy = float(np.sum(fold_test_y_pred == fold_test_y)) /␣
 ↪len(fold_test_y_pred)
        k_to_accuracies[k].append(accuracy)

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

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

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
```

```
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

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



[ ]:
```python
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = k_choices[accuracies_mean.argmax()]

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

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print(f"The best k values is {best_k}")
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
The best k values is 10
Got 141 / 500 correct => accuracy: 0.282000
```

Inline Question 3

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

$\mathit{Your Answer:}$ 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN.

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

$\mathit{Your Explanation:}$

1. The decision boundary of a k-NN classifier is non linear. The bounday depens on the arrangement of the training data.
2. The training error of a 1-NN will always be lower than or equal to that of 5-NN because the in 1-NN each point is classified as the label of the nearest traning point so generally training eror is very low. In 5-NN, the predicition considers 5 neighbours and the majority vote might sometimes misclassify the traning point leading to high training error.
3. The test error of 1-NN is often higher than that of 5-NN because 1-NN tends to overfit the training data. It captures noise and irregularities in the data, leading to worse generalization. In contrast, 5-NN smooths out the decision boundary, typically leading to better test performance by reducing overfitting.
4. The k-NN algorithm requires comparing the test example to every training example to compute distances, so the time complexity grows linearly with the size of the training set. Having more training points would lead more comparison which mens increasing the time required.

[ ]:
```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
```

```
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
The following additional packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono fonts-
texgyre
  fonts-urw-base35 libapache-pom-java libcmark-gfm-extensions0.29.0.gfm.3
libcmark-gfm0.29.0.gfm.3
  libcommons-logging-java libcommons-parent-java libfontbox-java libfontenc1
libgs9 libgs9-common
  libidn12 libijs-0.35 libjbig2dec0 libkpathsea6 libpdfbox-java libptexenc1
libruby3.0 libsynctex2
  libteckit0 libtexlua53 libtexluajit2 libwoff1 libzzip-0-13 lmodern pandoc-data
poppler-data
  preview-latex-style rake ruby ruby-net-telnet ruby-rubygems ruby-webrick ruby-
xmlrpc ruby3.0
  rubygems-integration t1utils teckit tex-common tex-gyre texlive-base texlive-
binaries
  texlive-fonts-recommended texlive-latex-base texlive-latex-recommended
texlive-pictures
  texlive-plain-generic tipa xfonts-encodings xfonts-utils
Suggested packages:
  fonts-noto fonts-freefont-otf | fonts-freefont-ttf libavalon-framework-java
  libcommons-logging-java-doc libexcalibur-logkit-java liblog4j1.2-java texlive-
luatex
  pandoc-citeproc context wkhtmltopdf librsvg2-bin groff ghc nodejs php python
libjs-mathjax
  libjs-katex citation-style-language-styles poppler-utils ghostscript fonts-
japanese-mincho
  | fonts-ipafont-mincho fonts-japanese-gothic | fonts-ipafont-gothic fonts-
arphic-ukai
  fonts-arphic-uming fonts-nanum ri ruby-dev bundler debhelper gv | postscript-
viewer perl-tk xpdf
  | pdf-viewer xzdec texlive-fonts-recommended-doc texlive-latex-base-doc
python3-pygments
  icc-profiles libfile-which-perl libspreadsheet-parseexcel-perl texlive-latex-
extra-doc
  texlive-latex-recommended-doc texlive-pstricks dot2tex prerex texlive-
pictures-doc vprerex
  default-jre-headless tipa-doc
The following NEW packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono fonts-
texgyre
  fonts-urw-base35 libapache-pom-java libcmark-gfm-extensions0.29.0.gfm.3
libcmark-gfm0.29.0.gfm.3
  libcommons-logging-java libcommons-parent-java libfontbox-java libfontenc1
libgs9 libgs9-common
  libidn12 libijs-0.35 libjbig2dec0 libkpathsea6 libpdfbox-java libptexenc1
```

```
libruby3.0 libsynctex2
  libteckit0 libtexlua53 libtexluajit2 libwoff1 libzzip-0-13 lmodern pandoc
pandoc-data
  poppler-data preview-latex-style rake ruby ruby-net-telnet ruby-rubygems ruby-
webrick ruby-xmlrpc
  ruby3.0 rubygems-integration t1utils teckit tex-common tex-gyre texlive
texlive-base
  texlive-binaries texlive-fonts-recommended texlive-latex-base texlive-latex-
extra
  texlive-latex-recommended texlive-pictures texlive-plain-generic texlive-xetex
tipa
  xfonts-encodings xfonts-utils
0 upgraded, 59 newly installed, 0 to remove and 49 not upgraded.
Need to get 202 MB of archives.
After this operation, 728 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-droid-fallback all
1:6.0.1r16-1.1build1 [1,805 kB]
Get:2 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-lato all 2.0-2.1
[2,696 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy/main amd64 poppler-data all
0.4.11-1 [2,171 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy/universe amd64 tex-common all 6.17
[33.7 kB]
Get:5 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-urw-base35 all
20200910-1 [6,367 kB]
Get:6 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libgs9-common
all 9.55.0~dfsg1-0ubuntu5.9 [752 kB]
Get:7 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libidn12 amd64
1.38-4ubuntu1 [60.0 kB]
Get:8 http://archive.ubuntu.com/ubuntu jammy/main amd64 libijs-0.35 amd64
0.35-15build2 [16.5 kB]
Get:9 http://archive.ubuntu.com/ubuntu jammy/main amd64 libjbig2dec0 amd64
0.19-3build2 [64.7 kB]
Get:10 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libgs9 amd64
9.55.0~dfsg1-0ubuntu5.9 [5,033 kB]
Get:11 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libkpathsea6
amd64 2021.20210626.59705-1ubuntu0.2 [60.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu jammy/main amd64 libwoff1 amd64
1.0.2-1build4 [45.2 kB]
Get:13 http://archive.ubuntu.com/ubuntu jammy/universe amd64 dvisvgm amd64
2.13.1-1 [1,221 kB]
Get:14 http://archive.ubuntu.com/ubuntu jammy/universe amd64 fonts-lmodern all
2.004.5-6.1 [4,532 kB]
Get:15 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-noto-mono all
20201225-1build1 [397 kB]
Get:16 http://archive.ubuntu.com/ubuntu jammy/universe amd64 fonts-texgyre all
20180621-3.1 [10.2 MB]
Get:17 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libapache-pom-java
```

```
all 18-1 [4,720 B]
Get:18 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcmark-
gfm0.29.0.gfm.3 amd64 0.29.0.gfm.3-3 [115 kB]
Get:19 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcmark-gfm-
extensions0.29.0.gfm.3 amd64 0.29.0.gfm.3-3 [25.1 kB]
Get:20 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcommons-parent-
java all 43-1 [10.8 kB]
Get:21 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcommons-logging-
java all 1.2-2 [60.3 kB]
Get:22 http://archive.ubuntu.com/ubuntu jammy/main amd64 libfontenc1 amd64
1:1.1.4-1build3 [14.7 kB]
Get:23 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libptexenc1
amd64 2021.20210626.59705-1ubuntu0.2 [39.1 kB]
Get:24 http://archive.ubuntu.com/ubuntu jammy/main amd64 rubygems-integration
all 1.18 [5,336 B]
Get:25 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 ruby3.0 amd64
3.0.2-7ubuntu2.7 [50.1 kB]
Get:26 http://archive.ubuntu.com/ubuntu jammy/main amd64 ruby-rubygems all
3.3.5-2 [228 kB]
Get:27 http://archive.ubuntu.com/ubuntu jammy/main amd64 ruby amd64 1:3.0~exp1
[5,100 B]
Get:28 http://archive.ubuntu.com/ubuntu jammy/main amd64 rake all 13.0.6-2 [61.7
kB]
Get:29 http://archive.ubuntu.com/ubuntu jammy/main amd64 ruby-net-telnet all
0.1.1-2 [12.6 kB]
Get:30 http://archive.ubuntu.com/ubuntu jammy/universe amd64 ruby-webrick all
1.7.0-3 [51.8 kB]
Get:31 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 ruby-xmlrpc all
0.3.2-1ubuntu0.1 [24.9 kB]
Get:32 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libruby3.0
amd64 3.0.2-7ubuntu2.7 [5,113 kB]
Get:33 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libsynctex2
amd64 2021.20210626.59705-1ubuntu0.2 [55.6 kB]
Get:34 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libteckit0 amd64
2.5.11+ds1-1 [421 kB]
Get:35 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libtexlua53
amd64 2021.20210626.59705-1ubuntu0.2 [120 kB]
Get:36 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libtexluajit2
amd64 2021.20210626.59705-1ubuntu0.2 [267 kB]
Get:37 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libzzip-0-13 amd64
0.13.72+dfsg.1-1.1 [27.0 kB]
Get:38 http://archive.ubuntu.com/ubuntu jammy/main amd64 xfonts-encodings all
1:1.0.5-0ubuntu2 [578 kB]
Get:39 http://archive.ubuntu.com/ubuntu jammy/main amd64 xfonts-utils amd64
1:7.7+6build2 [94.6 kB]
Get:40 http://archive.ubuntu.com/ubuntu jammy/universe amd64 lmodern all
2.004.5-6.1 [9,471 kB]
Get:41 http://archive.ubuntu.com/ubuntu jammy/universe amd64 pandoc-data all
```

```
2.9.2.1-3ubuntu2 [81.8 kB]
Get:42 http://archive.ubuntu.com/ubuntu jammy/universe amd64 pandoc amd64
2.9.2.1-3ubuntu2 [20.3 MB]
Get:43 http://archive.ubuntu.com/ubuntu jammy/universe amd64 preview-latex-style
all 12.2-1ubuntu1 [185 kB]
Get:44 http://archive.ubuntu.com/ubuntu jammy/main amd64 t1utils amd64
1.41-4build2 [61.3 kB]
Get:45 http://archive.ubuntu.com/ubuntu jammy/universe amd64 teckit amd64
2.5.11+ds1-1 [699 kB]
Get:46 http://archive.ubuntu.com/ubuntu jammy/universe amd64 tex-gyre all
20180621-3.1 [6,209 kB]
Get:47 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 texlive-
binaries amd64 2021.20210626.59705-1ubuntu0.2 [9,860 kB]
Get:48 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-base all
2021.20220204-1 [21.0 MB]
Get:49 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-fonts-
recommended all 2021.20220204-1 [4,972 kB]
Get:50 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-latex-base
all 2021.20220204-1 [1,128 kB]
Get:51 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-latex-
recommended all 2021.20220204-1 [14.4 MB]
Get:52 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive all
2021.20220204-1 [14.3 kB]
Get:53 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libfontbox-java all
1:1.8.16-2 [207 kB]
Get:54 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libpdfbox-java all
1:1.8.16-2 [5,199 kB]
Get:55 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-pictures
all 2021.20220204-1 [8,720 kB]
Get:56 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-latex-extra
all 2021.20220204-1 [13.9 MB]
Get:57 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-plain-
generic all 2021.20220204-1 [27.5 MB]
Get:58 http://archive.ubuntu.com/ubuntu jammy/universe amd64 tipa all 2:1.3-21
[2,967 kB]
Get:59 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-xetex all
2021.20220204-1 [12.4 MB]
Fetched 202 MB in 8s (26.4 MB/s)
Extracting templates from packages: 100%
Preconfiguring packages …
Selecting previously unselected package fonts-droid-fallback.
(Reading database … 123597 files and directories currently installed.)
Preparing to unpack …/00-fonts-droid-fallback_1%3a6.0.1r16-1.1build1_all.deb
…
Unpacking fonts-droid-fallback (1:6.0.1r16-1.1build1) …
Selecting previously unselected package fonts-lato.
Preparing to unpack …/01-fonts-lato_2.0-2.1_all.deb …
Unpacking fonts-lato (2.0-2.1) …
```

```
Selecting previously unselected package poppler-data.
Preparing to unpack …/02-poppler-data_0.4.11-1_all.deb …
Unpacking poppler-data (0.4.11-1) …
Selecting previously unselected package tex-common.
Preparing to unpack …/03-tex-common_6.17_all.deb …
Unpacking tex-common (6.17) …
Selecting previously unselected package fonts-urw-base35.
Preparing to unpack …/04-fonts-urw-base35_20200910-1_all.deb …
Unpacking fonts-urw-base35 (20200910-1) …
Selecting previously unselected package libgs9-common.
Preparing to unpack …/05-libgs9-common_9.55.0~dfsg1-0ubuntu5.9_all.deb …
Unpacking libgs9-common (9.55.0~dfsg1-0ubuntu5.9) …
Selecting previously unselected package libidn12:amd64.
Preparing to unpack …/06-libidn12_1.38-4ubuntu1_amd64.deb …
Unpacking libidn12:amd64 (1.38-4ubuntu1) …
Selecting previously unselected package libijs-0.35:amd64.
Preparing to unpack …/07-libijs-0.35_0.35-15build2_amd64.deb …
Unpacking libijs-0.35:amd64 (0.35-15build2) …
Selecting previously unselected package libjbig2dec0:amd64.
Preparing to unpack …/08-libjbig2dec0_0.19-3build2_amd64.deb …
Unpacking libjbig2dec0:amd64 (0.19-3build2) …
Selecting previously unselected package libgs9:amd64.
Preparing to unpack …/09-libgs9_9.55.0~dfsg1-0ubuntu5.9_amd64.deb …
Unpacking libgs9:amd64 (9.55.0~dfsg1-0ubuntu5.9) …
Selecting previously unselected package libkpathsea6:amd64.
Preparing to unpack …/10-libkpathsea6_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libkpathsea6:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libwoff1:amd64.
Preparing to unpack …/11-libwoff1_1.0.2-1build4_amd64.deb …
Unpacking libwoff1:amd64 (1.0.2-1build4) …
Selecting previously unselected package dvisvgm.
Preparing to unpack …/12-dvisvgm_2.13.1-1_amd64.deb …
Unpacking dvisvgm (2.13.1-1) …
Selecting previously unselected package fonts-lmodern.
Preparing to unpack …/13-fonts-lmodern_2.004.5-6.1_all.deb …
Unpacking fonts-lmodern (2.004.5-6.1) …
Selecting previously unselected package fonts-noto-mono.
Preparing to unpack …/14-fonts-noto-mono_20201225-1build1_all.deb …
Unpacking fonts-noto-mono (20201225-1build1) …
Selecting previously unselected package fonts-texgyre.
Preparing to unpack …/15-fonts-texgyre_20180621-3.1_all.deb …
Unpacking fonts-texgyre (20180621-3.1) …
Selecting previously unselected package libapache-pom-java.
Preparing to unpack …/16-libapache-pom-java_18-1_all.deb …
Unpacking libapache-pom-java (18-1) …
Selecting previously unselected package libcmark-gfm0.29.0.gfm.3:amd64.
Preparing to unpack …/17-libcmark-gfm0.29.0.gfm.3_0.29.0.gfm.3-3_amd64.deb …
```

```
Unpacking libcmark-gfm0.29.0.gfm.3:amd64 (0.29.0.gfm.3-3) …
Selecting previously unselected package libcmark-gfm-
extensions0.29.0.gfm.3:amd64.
Preparing to unpack …/18-libcmark-gfm-
extensions0.29.0.gfm.3_0.29.0.gfm.3-3_amd64.deb …
Unpacking libcmark-gfm-extensions0.29.0.gfm.3:amd64 (0.29.0.gfm.3-3) …
Selecting previously unselected package libcommons-parent-java.
Preparing to unpack …/19-libcommons-parent-java_43-1_all.deb …
Unpacking libcommons-parent-java (43-1) …
Selecting previously unselected package libcommons-logging-java.
Preparing to unpack …/20-libcommons-logging-java_1.2-2_all.deb …
Unpacking libcommons-logging-java (1.2-2) …
Selecting previously unselected package libfontenc1:amd64.
Preparing to unpack …/21-libfontenc1_1%3a1.1.4-1build3_amd64.deb …
Unpacking libfontenc1:amd64 (1:1.1.4-1build3) …
Selecting previously unselected package libptexenc1:amd64.
Preparing to unpack …/22-libptexenc1_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libptexenc1:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package rubygems-integration.
Preparing to unpack …/23-rubygems-integration_1.18_all.deb …
Unpacking rubygems-integration (1.18) …
Selecting previously unselected package ruby3.0.
Preparing to unpack …/24-ruby3.0_3.0.2-7ubuntu2.7_amd64.deb …
Unpacking ruby3.0 (3.0.2-7ubuntu2.7) …
Selecting previously unselected package ruby-rubygems.
Preparing to unpack …/25-ruby-rubygems_3.3.5-2_all.deb …
Unpacking ruby-rubygems (3.3.5-2) …
Selecting previously unselected package ruby.
Preparing to unpack …/26-ruby_1%3a3.0~exp1_amd64.deb …
Unpacking ruby (1:3.0~exp1) …
Selecting previously unselected package rake.
Preparing to unpack …/27-rake_13.0.6-2_all.deb …
Unpacking rake (13.0.6-2) …
Selecting previously unselected package ruby-net-telnet.
Preparing to unpack …/28-ruby-net-telnet_0.1.1-2_all.deb …
Unpacking ruby-net-telnet (0.1.1-2) …
Selecting previously unselected package ruby-webrick.
Preparing to unpack …/29-ruby-webrick_1.7.0-3_all.deb …
Unpacking ruby-webrick (1.7.0-3) …
Selecting previously unselected package ruby-xmlrpc.
Preparing to unpack …/30-ruby-xmlrpc_0.3.2-1ubuntu0.1_all.deb …
Unpacking ruby-xmlrpc (0.3.2-1ubuntu0.1) …
Selecting previously unselected package libruby3.0:amd64.
Preparing to unpack …/31-libruby3.0_3.0.2-7ubuntu2.7_amd64.deb …
Unpacking libruby3.0:amd64 (3.0.2-7ubuntu2.7) …
Selecting previously unselected package libsynctex2:amd64.
Preparing to unpack …/32-libsynctex2_2021.20210626.59705-1ubuntu0.2_amd64.deb
```

```
…
Unpacking libsynctex2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libteckit0:amd64.
Preparing to unpack …/33-libteckit0_2.5.11+ds1-1_amd64.deb …
Unpacking libteckit0:amd64 (2.5.11+ds1-1) …
Selecting previously unselected package libtexlua53:amd64.
Preparing to unpack …/34-libtexlua53_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libtexlua53:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libtexluajit2:amd64.
Preparing to unpack
…/35-libtexluajit2_2021.20210626.59705-1ubuntu0.2_amd64.deb …
Unpacking libtexluajit2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libzzip-0-13:amd64.
Preparing to unpack …/36-libzzip-0-13_0.13.72+dfsg.1-1.1_amd64.deb …
Unpacking libzzip-0-13:amd64 (0.13.72+dfsg.1-1.1) …
Selecting previously unselected package xfonts-encodings.
Preparing to unpack …/37-xfonts-encodings_1%3a1.0.5-0ubuntu2_all.deb …
Unpacking xfonts-encodings (1:1.0.5-0ubuntu2) …
Selecting previously unselected package xfonts-utils.
Preparing to unpack …/38-xfonts-utils_1%3a7.7+6build2_amd64.deb …
Unpacking xfonts-utils (1:7.7+6build2) …
Selecting previously unselected package lmodern.
Preparing to unpack …/39-lmodern_2.004.5-6.1_all.deb …
Unpacking lmodern (2.004.5-6.1) …
Selecting previously unselected package pandoc-data.
Preparing to unpack …/40-pandoc-data_2.9.2.1-3ubuntu2_all.deb …
Unpacking pandoc-data (2.9.2.1-3ubuntu2) …
Selecting previously unselected package pandoc.
Preparing to unpack …/41-pandoc_2.9.2.1-3ubuntu2_amd64.deb …
Unpacking pandoc (2.9.2.1-3ubuntu2) …
Selecting previously unselected package preview-latex-style.
Preparing to unpack …/42-preview-latex-style_12.2-1ubuntu1_all.deb …
Unpacking preview-latex-style (12.2-1ubuntu1) …
Selecting previously unselected package t1utils.
Preparing to unpack …/43-t1utils_1.41-4build2_amd64.deb …
Unpacking t1utils (1.41-4build2) …
Selecting previously unselected package teckit.
Preparing to unpack …/44-teckit_2.5.11+ds1-1_amd64.deb …
Unpacking teckit (2.5.11+ds1-1) …
Selecting previously unselected package tex-gyre.
Preparing to unpack …/45-tex-gyre_20180621-3.1_all.deb …
Unpacking tex-gyre (20180621-3.1) …
Selecting previously unselected package texlive-binaries.
Preparing to unpack …/46-texlive-
binaries_2021.20210626.59705-1ubuntu0.2_amd64.deb …
Unpacking texlive-binaries (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package texlive-base.
```

```
Preparing to unpack …/47-texlive-base_2021.20220204-1_all.deb …
Unpacking texlive-base (2021.20220204-1) …
Selecting previously unselected package texlive-fonts-recommended.
Preparing to unpack …/48-texlive-fonts-recommended_2021.20220204-1_all.deb …
Unpacking texlive-fonts-recommended (2021.20220204-1) …
Selecting previously unselected package texlive-latex-base.
Preparing to unpack …/49-texlive-latex-base_2021.20220204-1_all.deb …
Unpacking texlive-latex-base (2021.20220204-1) …
Selecting previously unselected package texlive-latex-recommended.
Preparing to unpack …/50-texlive-latex-recommended_2021.20220204-1_all.deb …
Unpacking texlive-latex-recommended (2021.20220204-1) …
Selecting previously unselected package texlive.
Preparing to unpack …/51-texlive_2021.20220204-1_all.deb …
Unpacking texlive (2021.20220204-1) …
Selecting previously unselected package libfontbox-java.
Preparing to unpack …/52-libfontbox-java_1%3a1.8.16-2_all.deb …
Unpacking libfontbox-java (1:1.8.16-2) …
Selecting previously unselected package libpdfbox-java.
Preparing to unpack …/53-libpdfbox-java_1%3a1.8.16-2_all.deb …
Unpacking libpdfbox-java (1:1.8.16-2) …
Selecting previously unselected package texlive-pictures.
Preparing to unpack …/54-texlive-pictures_2021.20220204-1_all.deb …
Unpacking texlive-pictures (2021.20220204-1) …
Selecting previously unselected package texlive-latex-extra.
Preparing to unpack …/55-texlive-latex-extra_2021.20220204-1_all.deb …
Unpacking texlive-latex-extra (2021.20220204-1) …
Selecting previously unselected package texlive-plain-generic.
Preparing to unpack …/56-texlive-plain-generic_2021.20220204-1_all.deb …
Unpacking texlive-plain-generic (2021.20220204-1) …
Selecting previously unselected package tipa.
Preparing to unpack …/57-tipa_2%3a1.3-21_all.deb …
Unpacking tipa (2:1.3-21) …
Selecting previously unselected package texlive-xetex.
Preparing to unpack …/58-texlive-xetex_2021.20220204-1_all.deb …
Unpacking texlive-xetex (2021.20220204-1) …
Setting up fonts-lato (2.0-2.1) …
Setting up fonts-noto-mono (20201225-1build1) …
Setting up libwoff1:amd64 (1.0.2-1build4) …
Setting up libtexlua53:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up libijs-0.35:amd64 (0.35-15build2) …
Setting up libtexluajit2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up libfontbox-java (1:1.8.16-2) …
Setting up rubygems-integration (1.18) …
Setting up libzzip-0-13:amd64 (0.13.72+dfsg.1-1.1) …
Setting up fonts-urw-base35 (20200910-1) …
Setting up poppler-data (0.4.11-1) …
Setting up tex-common (6.17) …
update-language: texlive-base not installed and configured, doing nothing!
```

```
Setting up libfontenc1:amd64 (1:1.1.4-1build3) …
Setting up libjbig2dec0:amd64 (0.19-3build2) …
Setting up libteckit0:amd64 (2.5.11+ds1-1) …
Setting up libapache-pom-java (18-1) …
Setting up ruby-net-telnet (0.1.1-2) …
Setting up xfonts-encodings (1:1.0.5-0ubuntu2) …
Setting up t1utils (1.41-4build2) …
Setting up libidn12:amd64 (1.38-4ubuntu1) …
Setting up fonts-texgyre (20180621-3.1) …
Setting up libkpathsea6:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up ruby-webrick (1.7.0-3) …
Setting up libcmark-gfm0.29.0.gfm.3:amd64 (0.29.0.gfm.3-3) …
Setting up fonts-lmodern (2.004.5-6.1) …
Setting up libcmark-gfm-extensions0.29.0.gfm.3:amd64 (0.29.0.gfm.3-3) …
Setting up fonts-droid-fallback (1:6.0.1r16-1.1build1) …
Setting up pandoc-data (2.9.2.1-3ubuntu2) …
Setting up ruby-xmlrpc (0.3.2-1ubuntu0.1) …
Setting up libsynctex2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up libgs9-common (9.55.0~dfsg1-0ubuntu5.9) …
Setting up teckit (2.5.11+ds1-1) …
Setting up libpdfbox-java (1:1.8.16-2) …
Setting up libgs9:amd64 (9.55.0~dfsg1-0ubuntu5.9) …
Setting up preview-latex-style (12.2-1ubuntu1) …
Setting up libcommons-parent-java (43-1) …
Setting up dvisvgm (2.13.1-1) …
Setting up libcommons-logging-java (1.2-2) …
Setting up xfonts-utils (1:7.7+6build2) …
Setting up libptexenc1:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up pandoc (2.9.2.1-3ubuntu2) …
Setting up texlive-binaries (2021.20210626.59705-1ubuntu0.2) …
update-alternatives: using /usr/bin/xdvi-xaw to provide /usr/bin/xdvi.bin
(xdvi.bin) in auto mode
update-alternatives: using /usr/bin/bibtex.original to provide /usr/bin/bibtex
(bibtex) in auto mode
Setting up lmodern (2.004.5-6.1) …
Setting up texlive-base (2021.20220204-1) …
/usr/bin/ucfr
/usr/bin/ucfr
/usr/bin/ucfr
/usr/bin/ucfr
mktexlsr: Updating /var/lib/texmf/ls-R-TEXLIVEDIST…
mktexlsr: Updating /var/lib/texmf/ls-R-TEXMFMAIN…
mktexlsr: Updating /var/lib/texmf/ls-R…
mktexlsr: Done.
tl-paper: setting paper size for dvips to a4:
/var/lib/texmf/dvips/config/config-paper.ps
tl-paper: setting paper size for dvipdfmx to a4:
/var/lib/texmf/dvipdfmx/dvipdfmx-paper.cfg
```

```
tl-paper: setting paper size for xdvi to a4: /var/lib/texmf/xdvi/XDvi-paper
tl-paper: setting paper size for pdftex to a4: /var/lib/texmf/tex/generic/tex-
ini-files/pdftexconfig.tex
Setting up tex-gyre (20180621-3.1) …
Setting up texlive-plain-generic (2021.20220204-1) …
Setting up texlive-latex-base (2021.20220204-1) …
Setting up texlive-latex-recommended (2021.20220204-1) …
Setting up texlive-pictures (2021.20220204-1) …
Setting up texlive-fonts-recommended (2021.20220204-1) …
Setting up tipa (2:1.3-21) …
Setting up texlive (2021.20220204-1) …
Setting up texlive-latex-extra (2021.20220204-1) …
Setting up texlive-xetex (2021.20220204-1) …
Setting up rake (13.0.6-2) …
Setting up libruby3.0:amd64 (3.0.2-7ubuntu2.7) …
Setting up ruby3.0 (3.0.2-7ubuntu2.7) …
Setting up ruby (1:3.0~exp1) …
Setting up ruby-rubygems (3.3.5-2) …
Processing triggers for man-db (2.10.2-1) …
Processing triggers for fontconfig (2.13.1-4.2ubuntu5) …
Processing triggers for libc-bin (2.35-0ubuntu3.4) …
/sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_5.so.3 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbb.so.12 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libur_adapter_opencl.so.0 is not a symbolic
link

/sbin/ldconfig.real: /usr/local/lib/libtbbmalloc.so.2 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_0.so.3 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbbind.so.3 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libur_adapter_level_zero.so.0 is not a
symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbmalloc_proxy.so.2 is not a symbolic
link

/sbin/ldconfig.real: /usr/local/lib/libur_loader.so.0 is not a symbolic link

Processing triggers for tex-common (6.17) …
Running updmap-sys. This may take some time… done.
Running mktexlsr /var/lib/texmf … done.
Building format(s) --all.
        This may take some time…
```

```
[ ]:
```

svm

September 11, 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
```

```
Mounted at /content/drive
/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

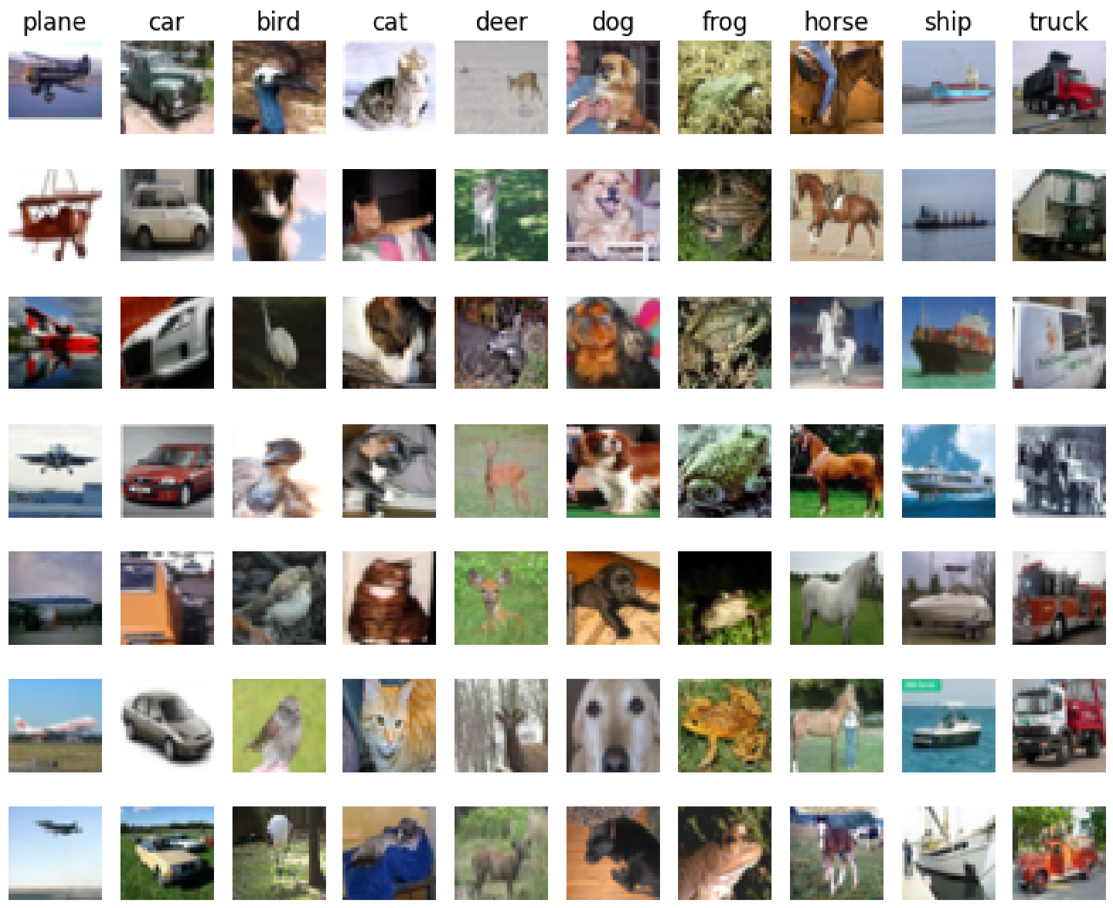
[ ]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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
```

1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

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



[ ]:
```
# 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)
```

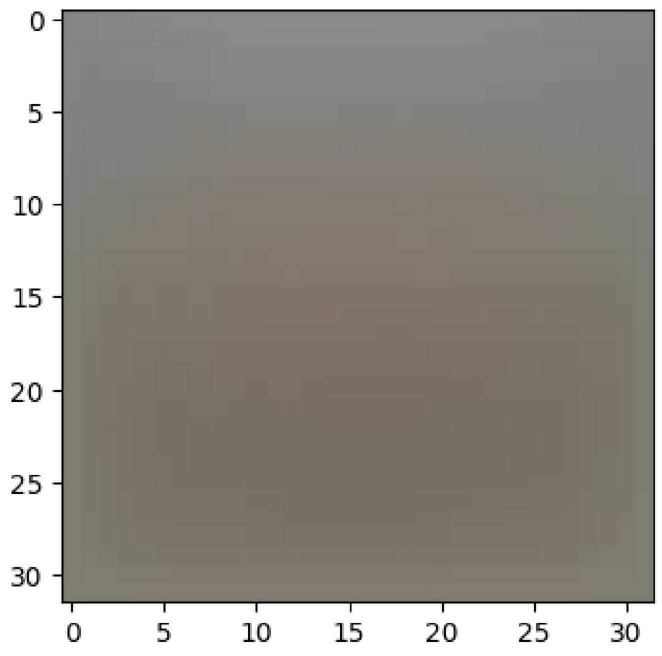
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

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

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

[ ]:
```python
# 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: 9.531079
```

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:

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

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

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

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

```
numerical: -5.032512 analytic: -5.032512, relative error: 1.020304e-11
numerical: -2.964459 analytic: -2.964459, relative error: 1.608079e-10
numerical: -10.775142 analytic: -10.775142, relative error: 1.075485e-11
numerical: 25.517231 analytic: 25.517231, relative error: 1.445577e-11
numerical: -0.892094 analytic: -0.892094, relative error: 4.129582e-10
numerical: -19.758774 analytic: -19.753552, relative error: 1.321423e-04
numerical: -3.484861 analytic: -3.484861, relative error: 9.020379e-11
numerical: 12.178366 analytic: 12.178366, relative error: 3.182702e-13
numerical: -22.149913 analytic: -22.045823, relative error: 2.355202e-03
numerical: 13.913351 analytic: 13.945525, relative error: 1.154864e-03
numerical: 35.609687 analytic: 35.609687, relative error: 7.983649e-12
numerical: 1.941137 analytic: 2.050514, relative error: 2.740147e-02
numerical: -3.074491 analytic: -3.074491, relative error: 2.395171e-10
numerical: 8.744490 analytic: 8.744490, relative error: 1.118317e-11
numerical: -14.940095 analytic: -14.940095, relative error: 8.008180e-12
numerical: -23.479135 analytic: -23.380655, relative error: 2.101600e-03
numerical: 6.883325 analytic: 6.883325, relative error: 9.706339e-12
numerical: 16.133951 analytic: 16.133951, relative error: 9.322103e-12
numerical: 33.020766 analytic: 33.020766, relative error: 1.521822e-11
numerical: 2.320716 analytic: 2.320716, relative error: 1.284166e-10
```

Inline Question 1

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

$Your Answer:$

1. The SVM loss function isnt differentiatble when the margin equals 1. This will cause a kink in the loss function which will lead to small mismatches in gradient checks.
2. In a 1D case, if the margin between two classes is exactly 1 then the gradient can suddenly change which will cause the gradient check to fail.
3. Smaller margins reduce the chance of hitting the non differentiable point whereas larger margins increase the frequency of gradient check failures because they make hitting the kink more likely.
4. The occasional mismatch in gradient checks for the SVM loss function is caused by non-differentiable points at the margin threshold. While this is expected and is not a major concern.

[ ]:
```
# 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: 9.531079e+00 computed in 0.090903s
Vectorized loss: 9.531079e+00 computed in 0.011325s
difference: -0.000000
```

[ ]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
```

```
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.089764s
Vectorized loss and gradient: computed in 0.020787s
difference: 0.000000
```

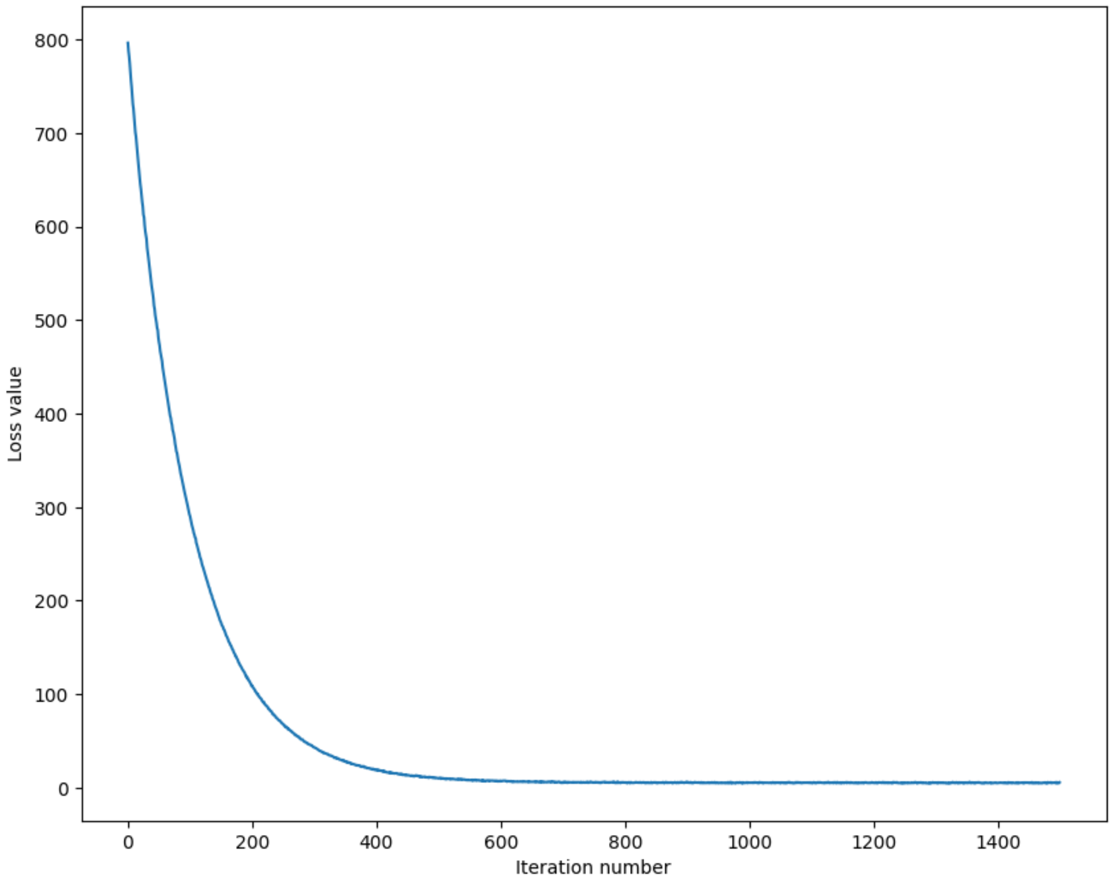
1.2.1 Stochastic Gradient Descent

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 796.420576
iteration 100 / 1500: loss 290.346676
iteration 200 / 1500: loss 109.149845
iteration 300 / 1500: loss 43.229903
iteration 400 / 1500: loss 19.044522
iteration 500 / 1500: loss 10.040602
iteration 600 / 1500: loss 6.890708
iteration 700 / 1500: loss 5.194942
iteration 800 / 1500: loss 5.814061
iteration 900 / 1500: loss 5.514196
iteration 1000 / 1500: loss 5.515129
iteration 1100 / 1500: loss 5.082229
iteration 1200 / 1500: loss 5.385040
iteration 1300 / 1500: loss 5.447317
iteration 1400 / 1500: loss 5.064766
That took 8.933474s
```

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



[22]:
```
# 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: 0.372265
validation accuracy: 0.380000
```

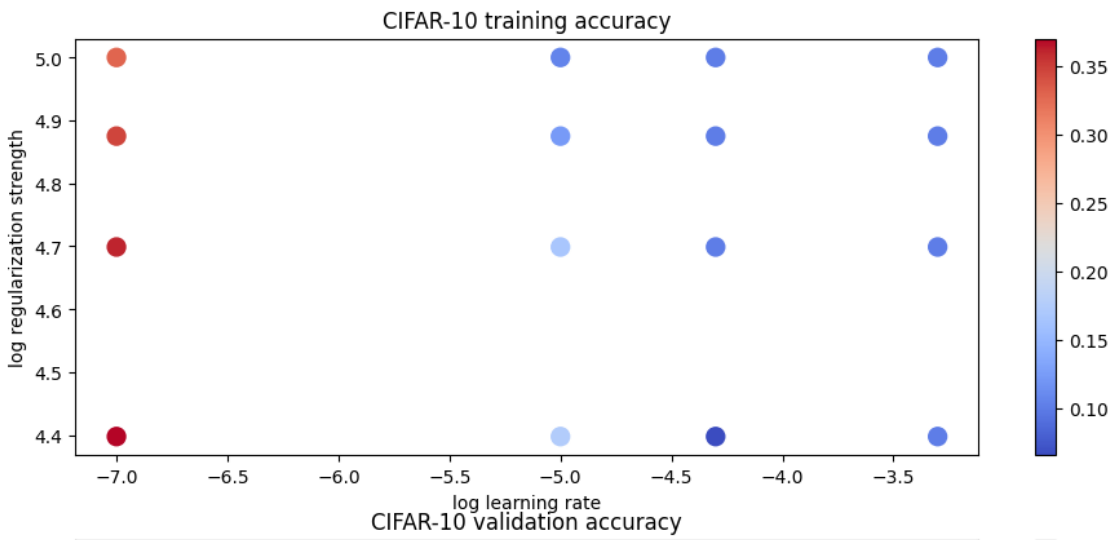
[29]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) on the validation set.
```

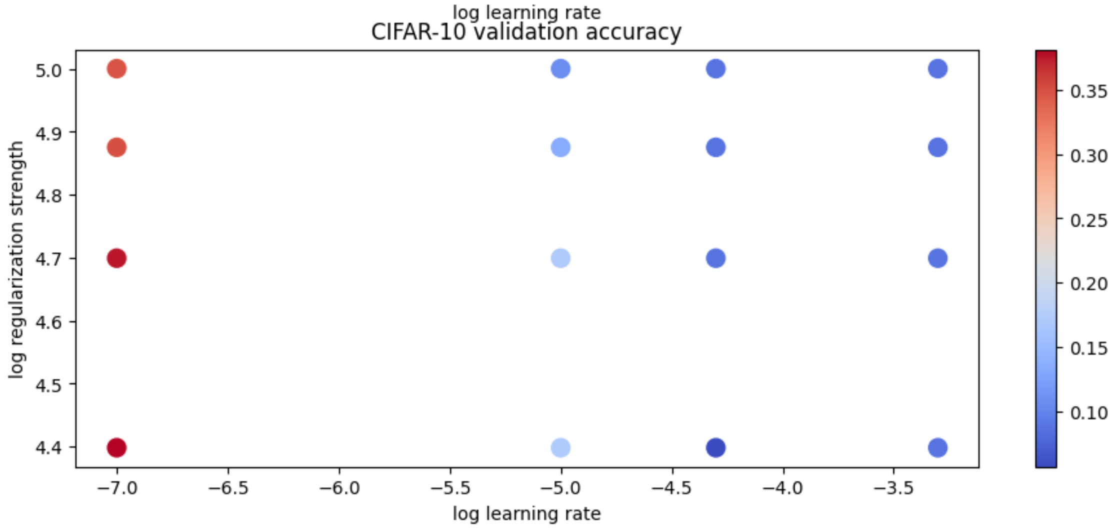
```python
# Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

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

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

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

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Define the possible hyperparameters to try
learning_rates = [1e-7,  5e-5, 1e-5, 5e-4]
regularization_strengths = [2.5e4, 5e4, 7.5e4, 1e5]

for lr in learning_rates:
  for reg in regularization_strengths:

    svm = LinearSVM()

    svm.train(X_train,y_train, learning_rate=lr, reg=reg, num_iters=1500)

    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)
    train_accuracy = np.mean(y_train_pred == y_train)
    val_accuracy = np.mean(y_val_pred == y_val)

    results[(lr, reg)] = (train_accuracy, val_accuracy)

    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm


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

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

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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.370082 val accuracy: 0.381000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.360857 val accuracy: 0.375000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.347265 val accuracy: 0.350000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.328898 val accuracy: 0.347000
lr 1.000000e-05 reg 2.500000e+04 train accuracy: 0.174306 val accuracy: 0.170000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.165633 val accuracy: 0.170000
lr 1.000000e-05 reg 7.500000e+04 train accuracy: 0.123490 val accuracy: 0.134000
lr 1.000000e-05 reg 1.000000e+05 train accuracy: 0.105286 val accuracy: 0.107000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.065776 val accuracy: 0.057000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 7.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-04 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-04 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-04 reg 7.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-04 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.381000
```

[30]:
```
# 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()
```



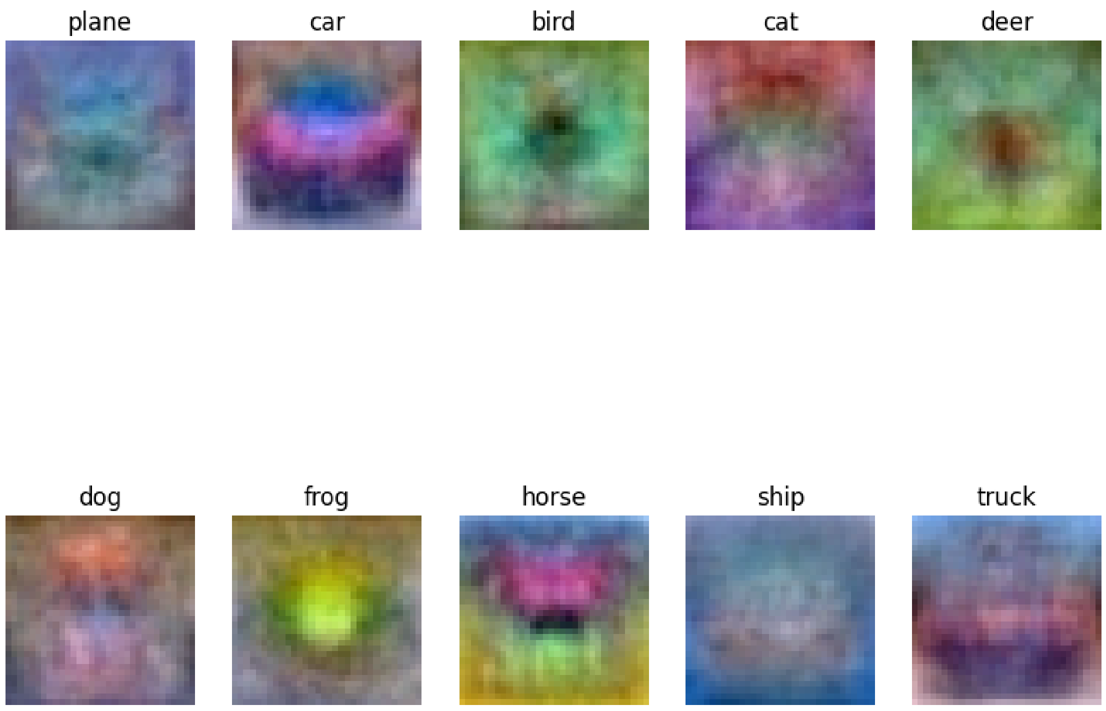


[31]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.361000
```

[32]:
```
# 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.

Your Answer :

1. The visualized SVM weights appear as blurry which is an abstract representations of each class which capturing general shapes and color patterns but lacking fine details. This is because SVMs are linear models that focus on broad, averaged features across the training data.

2. The colors and regions highlight the important features that the SVM learned for each class, with different colors corresponding to the dominant patterns or pixel intensities that help the classifier differentiate between classes.

3. The blurriness and abstraction occur because SVMs are linear classifiers which capture average patterns across many images of each class. They focus on general features like color

and rough shape but do not capture deep details or complex relationships, leading to these generalized, blurred visualizations.

softmax

September 11, 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
```

```
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

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

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

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

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

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

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

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

```
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

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

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

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


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

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

1.1 Softmax Classifier

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

[ ]:
```python
# 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: 2.380038
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$

1. The softmax converts the raw scores into probabilities for each class. If the model is not good i.e not trained well the probabilities of for all classes will be nearly equal.
2. Since we have a 10-class classification problem, each class's probablitiy will be equal to 1/10 which is 0.1.
3. The loss for the corrected class is calculated as the negative log of the predicted probability($-\log$(p correct))
4. If p(correct) is nearly to 0.1, the loss becomes -log(0.1) which is equal to 2.3

[8]:
```python
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

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

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

```
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.646235 analytic: -0.646235, relative error: 2.361992e-08
numerical: 2.925352 analytic: 2.925352, relative error: 1.255406e-08
numerical: -0.246280 analytic: -0.246280, relative error: 9.549454e-08
numerical: -3.613400 analytic: -3.613400, relative error: 2.091076e-09
numerical: -2.543145 analytic: -2.543145, relative error: 1.515000e-08
numerical: 0.498404 analytic: 0.498404, relative error: 1.995603e-07
numerical: -0.053435 analytic: -0.053435, relative error: 1.273070e-06
numerical: 2.668769 analytic: 2.668768, relative error: 2.904967e-08
numerical: -5.105478 analytic: -5.105478, relative error: 9.784430e-09
numerical: -1.009986 analytic: -1.009986, relative error: 2.041836e-08
numerical: 1.068759 analytic: 1.068759, relative error: 1.427443e-08
numerical: 0.688643 analytic: 0.688643, relative error: 4.151572e-09
numerical: 0.758017 analytic: 0.758016, relative error: 3.422164e-08
numerical: -0.382230 analytic: -0.382230, relative error: 5.356424e-08
numerical: 0.778870 analytic: 0.778870, relative error: 1.788551e-07
numerical: -1.123873 analytic: -1.123873, relative error: 1.526717e-08
numerical: 2.888895 analytic: 2.888895, relative error: 1.941091e-08
numerical: 3.358653 analytic: 3.358653, relative error: 8.697741e-09
numerical: 2.036284 analytic: 2.036284, relative error: 2.412014e-08
numerical: -4.842233 analytic: -4.842234, relative error: 1.580256e-08
```

[10]:
```
# 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: 2.380038e+00 computed in 0.101055s
vectorized loss: 2.380038e+00 computed in 0.013091s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[12]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

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

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

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



# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
  for reg in regularization_strengths:
    # Intialize a classifier
    softmax = Softmax()

    # Train the softmax
    loss_history= softmax.train(X_train, y_train,learning_rate=lr, reg=reg,␣
 ↪num_iters=1500, verbose=False)

    # Predicting on training and validation
    y_train_pred= softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)

    # Computing accuracy
    train_accuracy = np.mean(y_train == y_train_pred)
    val_accuracy = np.mean(y_val == y_val_pred)
    # store the results
    results[(lr, reg)] = (train_accuracy, val_accuracy)

    # If this is the best validation accuracy so far, store the model
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_softmax = softmax


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

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

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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.351449 val accuracy: 0.374000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.327388 val accuracy: 0.342000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.344592 val accuracy: 0.346000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.327816 val accuracy: 0.342000
best validation accuracy achieved during cross-validation: 0.374000
```

[13]:
```
# 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: 0.365000
```

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.

$\mathit{Your Answer:}$ True

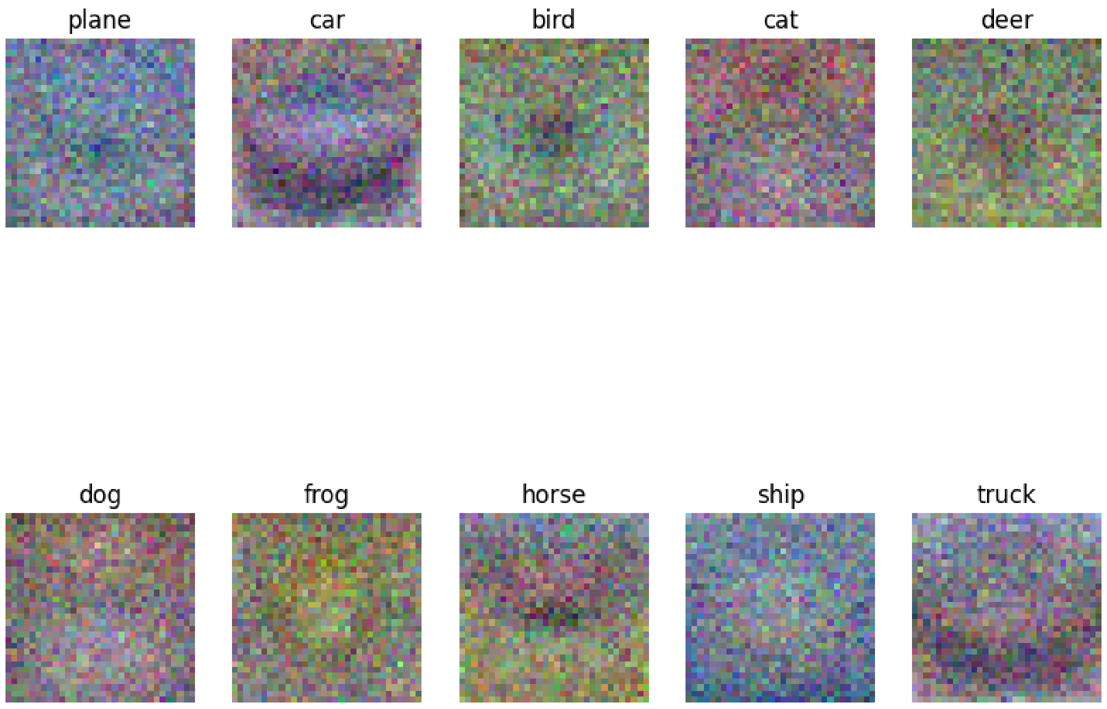
$\mathit{Your Explanation:}$

1. The hinge loss used in the SVM classifier only penalizes examples that violate the margin.
2. For any given training example, the SVM loss is only affected if the margin violation occurs, i.e., if **si - sy + 1 > 0**.
3. If the example is classified correctly and is far enough from the decision boundary (i.e., within the margin), adding this example does not affect the loss. So a new datapoint will not change

the loss if it is classified correctly and doesn't violate the margin.

4. The Softmax loss always takes into account the probability distribution over all classes.
5. Every datapoint contributes to the loss regardless of how well it is classified. Even if the correct class has a much higher score than the others the softmax loss will still be affected by adding a new datapoint because it adjusts the probabilities of all classes.
6. The only time it contributes no additional loss is when the probability of the correct class is exactly 1, which is practically impossible with real data

[14]:
```
# 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

September 11, 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:  9.769849468192957e-10
```

3 Affine layer: backward

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

[5]:
```python
# 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]:
```python
# 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:

1. Sigmoid
    - sigmoid function squeeze the inputs to range between 0,1
    - Example: For a really big +ve number or -ve number (x=10 or x=-10) the gradient becomes 0. This makes it makes learning process slow since the network stops adjusting weights properly.
2. ReLU
    - It keeps +ve values the same and -ve value to be zero.
    - For all the -ve inputs, the gradient inputs the gradient is zero. So the neurons will stop learning if it gets stuck in the negative area.

3. Leaky ReLU

- It works similar to ReLU the only differnce is that there is a small constant called alpha.
- When the inputs are -ve there's a still is a small gradient so there is some learning going on.

So both Sigmoid and ReLU suffer from zero gradients whereas Leaky ReLU does not

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.

[12]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

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

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

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

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

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

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

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.20e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

9 Solver

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

[18]:
```
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,                      # Pass the model
    data,                       # Pass the dataset
    optim_config={               # Optimizer config (e.g., learning rate)
        'learning_rate': 1e-3,
    }
)
solver.train()

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

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

```
(Iteration 1 / 4900) loss: 2.300699
(Epoch 0 / 10) train acc: 0.120000; val_acc: 0.126000
(Iteration 11 / 4900) loss: 2.269798
(Iteration 21 / 4900) loss: 2.173726
(Iteration 31 / 4900) loss: 2.085978
(Iteration 41 / 4900) loss: 2.046111
(Iteration 51 / 4900) loss: 1.856365
(Iteration 61 / 4900) loss: 1.869808
(Iteration 71 / 4900) loss: 2.042406
(Iteration 81 / 4900) loss: 1.880712
(Iteration 91 / 4900) loss: 1.938760
(Iteration 101 / 4900) loss: 1.910774
(Iteration 111 / 4900) loss: 1.870556
(Iteration 121 / 4900) loss: 1.714769
(Iteration 131 / 4900) loss: 1.654398
(Iteration 141 / 4900) loss: 1.785264
(Iteration 151 / 4900) loss: 1.842792
(Iteration 161 / 4900) loss: 1.715698
(Iteration 171 / 4900) loss: 1.788475
(Iteration 181 / 4900) loss: 1.898668
(Iteration 191 / 4900) loss: 1.709677
(Iteration 201 / 4900) loss: 1.805298
(Iteration 211 / 4900) loss: 1.763619
(Iteration 221 / 4900) loss: 1.756287
(Iteration 231 / 4900) loss: 1.691266
(Iteration 241 / 4900) loss: 1.801791
(Iteration 251 / 4900) loss: 1.643285
(Iteration 261 / 4900) loss: 1.748266
(Iteration 271 / 4900) loss: 1.634148
(Iteration 281 / 4900) loss: 1.574631
(Iteration 291 / 4900) loss: 1.684696
(Iteration 301 / 4900) loss: 1.712573
(Iteration 311 / 4900) loss: 1.642877
(Iteration 321 / 4900) loss: 1.733442
(Iteration 331 / 4900) loss: 1.906105
(Iteration 341 / 4900) loss: 1.535519
(Iteration 351 / 4900) loss: 1.727252
(Iteration 361 / 4900) loss: 1.502057
(Iteration 371 / 4900) loss: 1.727812
(Iteration 381 / 4900) loss: 1.744832
(Iteration 391 / 4900) loss: 1.624386
(Iteration 401 / 4900) loss: 1.572095
(Iteration 411 / 4900) loss: 1.790865
(Iteration 421 / 4900) loss: 1.673037
(Iteration 431 / 4900) loss: 1.667437
```

```
(Iteration 441 / 4900) loss: 1.645554
(Iteration 451 / 4900) loss: 1.654044
(Iteration 461 / 4900) loss: 1.468038
(Iteration 471 / 4900) loss: 1.576049
(Iteration 481 / 4900) loss: 1.406982
(Epoch 1 / 10) train acc: 0.417000; val_acc: 0.464000
(Iteration 491 / 4900) loss: 1.498469
(Iteration 501 / 4900) loss: 1.467151
(Iteration 511 / 4900) loss: 1.555981
(Iteration 521 / 4900) loss: 1.565861
(Iteration 531 / 4900) loss: 1.782464
(Iteration 541 / 4900) loss: 1.383583
(Iteration 551 / 4900) loss: 1.803421
(Iteration 561 / 4900) loss: 1.589169
(Iteration 571 / 4900) loss: 1.597800
(Iteration 581 / 4900) loss: 1.533993
(Iteration 591 / 4900) loss: 1.519559
(Iteration 601 / 4900) loss: 1.294213
(Iteration 611 / 4900) loss: 1.541863
(Iteration 621 / 4900) loss: 1.430180
(Iteration 631 / 4900) loss: 1.654597
(Iteration 641 / 4900) loss: 1.673291
(Iteration 651 / 4900) loss: 1.440074
(Iteration 661 / 4900) loss: 1.639858
(Iteration 671 / 4900) loss: 1.585000
(Iteration 681 / 4900) loss: 1.697306
(Iteration 691 / 4900) loss: 1.617519
(Iteration 701 / 4900) loss: 1.633937
(Iteration 711 / 4900) loss: 1.518262
(Iteration 721 / 4900) loss: 1.410813
(Iteration 731 / 4900) loss: 1.470912
(Iteration 741 / 4900) loss: 1.644008
(Iteration 751 / 4900) loss: 1.411575
(Iteration 761 / 4900) loss: 1.578529
(Iteration 771 / 4900) loss: 1.519405
(Iteration 781 / 4900) loss: 1.503312
(Iteration 791 / 4900) loss: 1.218550
(Iteration 801 / 4900) loss: 1.526950
(Iteration 811 / 4900) loss: 1.461846
(Iteration 821 / 4900) loss: 1.545994
(Iteration 831 / 4900) loss: 1.732973
(Iteration 841 / 4900) loss: 1.339148
(Iteration 851 / 4900) loss: 1.505104
(Iteration 861 / 4900) loss: 1.633667
(Iteration 871 / 4900) loss: 1.431872
(Iteration 881 / 4900) loss: 1.549745
(Iteration 891 / 4900) loss: 1.543462
(Iteration 901 / 4900) loss: 1.382463
```

```
(Iteration 911 / 4900) loss: 1.373037
(Iteration 921 / 4900) loss: 1.541711
(Iteration 931 / 4900) loss: 1.360641
(Iteration 941 / 4900) loss: 1.461141
(Iteration 951 / 4900) loss: 1.454621
(Iteration 961 / 4900) loss: 1.512184
(Iteration 971 / 4900) loss: 1.506731
(Epoch 2 / 10) train acc: 0.476000; val_acc: 0.446000
(Iteration 981 / 4900) loss: 1.619201
(Iteration 991 / 4900) loss: 1.421195
(Iteration 1001 / 4900) loss: 1.444711
(Iteration 1011 / 4900) loss: 1.440287
(Iteration 1021 / 4900) loss: 1.495290
(Iteration 1031 / 4900) loss: 1.369471
(Iteration 1041 / 4900) loss: 1.755257
(Iteration 1051 / 4900) loss: 1.477055
(Iteration 1061 / 4900) loss: 1.500866
(Iteration 1071 / 4900) loss: 1.469064
(Iteration 1081 / 4900) loss: 1.381221
(Iteration 1091 / 4900) loss: 1.469116
(Iteration 1101 / 4900) loss: 1.555706
(Iteration 1111 / 4900) loss: 1.441766
(Iteration 1121 / 4900) loss: 1.482924
(Iteration 1131 / 4900) loss: 1.441505
(Iteration 1141 / 4900) loss: 1.257500
(Iteration 1151 / 4900) loss: 1.592310
(Iteration 1161 / 4900) loss: 1.552527
(Iteration 1171 / 4900) loss: 1.546135
(Iteration 1181 / 4900) loss: 1.554858
(Iteration 1191 / 4900) loss: 1.407781
(Iteration 1201 / 4900) loss: 1.493467
(Iteration 1211 / 4900) loss: 1.570307
(Iteration 1221 / 4900) loss: 1.155740
(Iteration 1231 / 4900) loss: 1.301597
(Iteration 1241 / 4900) loss: 1.309929
(Iteration 1251 / 4900) loss: 1.323352
(Iteration 1261 / 4900) loss: 1.468657
(Iteration 1271 / 4900) loss: 1.499634
(Iteration 1281 / 4900) loss: 1.442742
(Iteration 1291 / 4900) loss: 1.371541
(Iteration 1301 / 4900) loss: 1.471069
(Iteration 1311 / 4900) loss: 1.482168
(Iteration 1321 / 4900) loss: 1.499757
(Iteration 1331 / 4900) loss: 1.636848
(Iteration 1341 / 4900) loss: 1.257517
(Iteration 1351 / 4900) loss: 1.517766
(Iteration 1361 / 4900) loss: 1.182979
(Iteration 1371 / 4900) loss: 1.501404
```

```
(Iteration 1381 / 4900) loss: 1.306282
(Iteration 1391 / 4900) loss: 1.619585
(Iteration 1401 / 4900) loss: 1.153571
(Iteration 1411 / 4900) loss: 1.266762
(Iteration 1421 / 4900) loss: 1.281106
(Iteration 1431 / 4900) loss: 1.328511
(Iteration 1441 / 4900) loss: 1.304726
(Iteration 1451 / 4900) loss: 1.452458
(Iteration 1461 / 4900) loss: 1.547329
(Epoch 3 / 10) train acc: 0.505000; val_acc: 0.445000
(Iteration 1471 / 4900) loss: 1.500409
(Iteration 1481 / 4900) loss: 1.441319
(Iteration 1491 / 4900) loss: 1.432092
(Iteration 1501 / 4900) loss: 1.718235
(Iteration 1511 / 4900) loss: 1.376164
(Iteration 1521 / 4900) loss: 1.346380
(Iteration 1531 / 4900) loss: 1.390763
(Iteration 1541 / 4900) loss: 1.597262
(Iteration 1551 / 4900) loss: 1.614772
(Iteration 1561 / 4900) loss: 1.326323
(Iteration 1571 / 4900) loss: 1.676069
(Iteration 1581 / 4900) loss: 1.516803
(Iteration 1591 / 4900) loss: 1.376019
(Iteration 1601 / 4900) loss: 1.352254
(Iteration 1611 / 4900) loss: 1.404193
(Iteration 1621 / 4900) loss: 1.539288
(Iteration 1631 / 4900) loss: 1.697553
(Iteration 1641 / 4900) loss: 1.427398
(Iteration 1651 / 4900) loss: 1.347777
(Iteration 1661 / 4900) loss: 1.458780
(Iteration 1671 / 4900) loss: 1.505840
(Iteration 1681 / 4900) loss: 1.532337
(Iteration 1691 / 4900) loss: 1.472812
(Iteration 1701 / 4900) loss: 1.385284
(Iteration 1711 / 4900) loss: 1.320631
(Iteration 1721 / 4900) loss: 1.576710
(Iteration 1731 / 4900) loss: 1.418836
(Iteration 1741 / 4900) loss: 1.444940
(Iteration 1751 / 4900) loss: 1.562110
(Iteration 1761 / 4900) loss: 1.430330
(Iteration 1771 / 4900) loss: 1.326339
(Iteration 1781 / 4900) loss: 1.495258
(Iteration 1791 / 4900) loss: 1.340686
(Iteration 1801 / 4900) loss: 1.357755
(Iteration 1811 / 4900) loss: 1.226755
(Iteration 1821 / 4900) loss: 1.532801
(Iteration 1831 / 4900) loss: 1.420135
(Iteration 1841 / 4900) loss: 1.225880
```

```
(Iteration 1851 / 4900) loss: 1.500234
(Iteration 1861 / 4900) loss: 1.253152
(Iteration 1871 / 4900) loss: 1.404135
(Iteration 1881 / 4900) loss: 1.191582
(Iteration 1891 / 4900) loss: 1.499622
(Iteration 1901 / 4900) loss: 1.230543
(Iteration 1911 / 4900) loss: 1.253432
(Iteration 1921 / 4900) loss: 1.495255
(Iteration 1931 / 4900) loss: 1.497087
(Iteration 1941 / 4900) loss: 1.405401
(Iteration 1951 / 4900) loss: 1.364168
(Epoch 4 / 10) train acc: 0.520000; val_acc: 0.474000
(Iteration 1961 / 4900) loss: 1.503698
(Iteration 1971 / 4900) loss: 1.273257
(Iteration 1981 / 4900) loss: 1.739895
(Iteration 1991 / 4900) loss: 1.312162
(Iteration 2001 / 4900) loss: 1.317045
(Iteration 2011 / 4900) loss: 1.473751
(Iteration 2021 / 4900) loss: 1.556969
(Iteration 2031 / 4900) loss: 1.350413
(Iteration 2041 / 4900) loss: 1.407603
(Iteration 2051 / 4900) loss: 1.335905
(Iteration 2061 / 4900) loss: 1.434221
(Iteration 2071 / 4900) loss: 1.513751
(Iteration 2081 / 4900) loss: 1.418429
(Iteration 2091 / 4900) loss: 1.461166
(Iteration 2101 / 4900) loss: 1.408614
(Iteration 2111 / 4900) loss: 1.116970
(Iteration 2121 / 4900) loss: 1.446613
(Iteration 2131 / 4900) loss: 1.310392
(Iteration 2141 / 4900) loss: 1.358767
(Iteration 2151 / 4900) loss: 1.355210
(Iteration 2161 / 4900) loss: 1.380946
(Iteration 2171 / 4900) loss: 1.402022
(Iteration 2181 / 4900) loss: 1.415037
(Iteration 2191 / 4900) loss: 1.454394
(Iteration 2201 / 4900) loss: 1.538331
(Iteration 2211 / 4900) loss: 1.280523
(Iteration 2221 / 4900) loss: 1.397050
(Iteration 2231 / 4900) loss: 1.214963
(Iteration 2241 / 4900) loss: 1.281021
(Iteration 2251 / 4900) loss: 1.410096
(Iteration 2261 / 4900) loss: 1.458806
(Iteration 2271 / 4900) loss: 1.431149
(Iteration 2281 / 4900) loss: 1.391702
(Iteration 2291 / 4900) loss: 1.301815
(Iteration 2301 / 4900) loss: 1.457151
(Iteration 2311 / 4900) loss: 1.455909
```

```
(Iteration 2321 / 4900) loss: 1.484610
(Iteration 2331 / 4900) loss: 1.388006
(Iteration 2341 / 4900) loss: 1.420509
(Iteration 2351 / 4900) loss: 1.569586
(Iteration 2361 / 4900) loss: 1.376179
(Iteration 2371 / 4900) loss: 1.280749
(Iteration 2381 / 4900) loss: 1.503704
(Iteration 2391 / 4900) loss: 1.434778
(Iteration 2401 / 4900) loss: 1.534981
(Iteration 2411 / 4900) loss: 1.445242
(Iteration 2421 / 4900) loss: 1.323393
(Iteration 2431 / 4900) loss: 1.534823
(Iteration 2441 / 4900) loss: 1.321399
(Epoch 5 / 10) train acc: 0.490000; val_acc: 0.432000
(Iteration 2451 / 4900) loss: 1.362420
(Iteration 2461 / 4900) loss: 1.189280
(Iteration 2471 / 4900) loss: 1.127819
(Iteration 2481 / 4900) loss: 1.431343
(Iteration 2491 / 4900) loss: 1.237619
(Iteration 2501 / 4900) loss: 1.361783
(Iteration 2511 / 4900) loss: 1.351230
(Iteration 2521 / 4900) loss: 1.410980
(Iteration 2531 / 4900) loss: 1.246118
(Iteration 2541 / 4900) loss: 1.307581
(Iteration 2551 / 4900) loss: 1.416507
(Iteration 2561 / 4900) loss: 1.361773
(Iteration 2571 / 4900) loss: 1.407474
(Iteration 2581 / 4900) loss: 1.581603
(Iteration 2591 / 4900) loss: 1.379032
(Iteration 2601 / 4900) loss: 1.530644
(Iteration 2611 / 4900) loss: 1.583547
(Iteration 2621 / 4900) loss: 1.242328
(Iteration 2631 / 4900) loss: 1.259323
(Iteration 2641 / 4900) loss: 1.252881
(Iteration 2651 / 4900) loss: 1.536233
(Iteration 2661 / 4900) loss: 1.230710
(Iteration 2671 / 4900) loss: 1.311979
(Iteration 2681 / 4900) loss: 1.528243
(Iteration 2691 / 4900) loss: 1.295718
(Iteration 2701 / 4900) loss: 1.440835
(Iteration 2711 / 4900) loss: 1.525200
(Iteration 2721 / 4900) loss: 1.279660
(Iteration 2731 / 4900) loss: 1.307616
(Iteration 2741 / 4900) loss: 1.341927
(Iteration 2751 / 4900) loss: 1.514432
(Iteration 2761 / 4900) loss: 1.172422
(Iteration 2771 / 4900) loss: 1.311129
(Iteration 2781 / 4900) loss: 1.336539
```

```
(Iteration 2791 / 4900) loss: 1.538915
(Iteration 2801 / 4900) loss: 1.260611
(Iteration 2811 / 4900) loss: 1.496698
(Iteration 2821 / 4900) loss: 1.214055
(Iteration 2831 / 4900) loss: 1.270842
(Iteration 2841 / 4900) loss: 1.406694
(Iteration 2851 / 4900) loss: 1.255938
(Iteration 2861 / 4900) loss: 1.440570
(Iteration 2871 / 4900) loss: 1.160745
(Iteration 2881 / 4900) loss: 1.247663
(Iteration 2891 / 4900) loss: 1.459794
(Iteration 2901 / 4900) loss: 1.351395
(Iteration 2911 / 4900) loss: 1.249169
(Iteration 2921 / 4900) loss: 1.141448
(Iteration 2931 / 4900) loss: 1.305115
(Epoch 6 / 10) train acc: 0.523000; val_acc: 0.487000
(Iteration 2941 / 4900) loss: 1.403042
(Iteration 2951 / 4900) loss: 1.504741
(Iteration 2961 / 4900) loss: 1.224883
(Iteration 2971 / 4900) loss: 1.410308
(Iteration 2981 / 4900) loss: 1.440153
(Iteration 2991 / 4900) loss: 1.398433
(Iteration 3001 / 4900) loss: 1.380496
(Iteration 3011 / 4900) loss: 1.377536
(Iteration 3021 / 4900) loss: 1.505669
(Iteration 3031 / 4900) loss: 1.422549
(Iteration 3041 / 4900) loss: 1.295843
(Iteration 3051 / 4900) loss: 1.537606
(Iteration 3061 / 4900) loss: 1.522665
(Iteration 3071 / 4900) loss: 1.522869
(Iteration 3081 / 4900) loss: 1.211363
(Iteration 3091 / 4900) loss: 1.247706
(Iteration 3101 / 4900) loss: 1.354334
(Iteration 3111 / 4900) loss: 1.535542
(Iteration 3121 / 4900) loss: 1.277506
(Iteration 3131 / 4900) loss: 1.286847
(Iteration 3141 / 4900) loss: 1.323073
(Iteration 3151 / 4900) loss: 1.266125
(Iteration 3161 / 4900) loss: 1.171099
(Iteration 3171 / 4900) loss: 1.456163
(Iteration 3181 / 4900) loss: 1.299184
(Iteration 3191 / 4900) loss: 1.571131
(Iteration 3201 / 4900) loss: 1.242490
(Iteration 3211 / 4900) loss: 1.406898
(Iteration 3221 / 4900) loss: 1.348645
(Iteration 3231 / 4900) loss: 1.267357
(Iteration 3241 / 4900) loss: 1.201660
(Iteration 3251 / 4900) loss: 1.285804
```

```
(Iteration 3261 / 4900) loss: 1.129579
(Iteration 3271 / 4900) loss: 1.326218
(Iteration 3281 / 4900) loss: 1.389818
(Iteration 3291 / 4900) loss: 1.334045
(Iteration 3301 / 4900) loss: 1.460675
(Iteration 3311 / 4900) loss: 1.461704
(Iteration 3321 / 4900) loss: 1.447291
(Iteration 3331 / 4900) loss: 1.183972
(Iteration 3341 / 4900) loss: 1.367404
(Iteration 3351 / 4900) loss: 1.205156
(Iteration 3361 / 4900) loss: 1.317654
(Iteration 3371 / 4900) loss: 1.250887
(Iteration 3381 / 4900) loss: 1.465826
(Iteration 3391 / 4900) loss: 1.334721
(Iteration 3401 / 4900) loss: 1.193784
(Iteration 3411 / 4900) loss: 1.480159
(Iteration 3421 / 4900) loss: 1.301977
(Epoch 7 / 10) train acc: 0.523000; val_acc: 0.460000
(Iteration 3431 / 4900) loss: 1.456996
(Iteration 3441 / 4900) loss: 1.238671
(Iteration 3451 / 4900) loss: 1.304855
(Iteration 3461 / 4900) loss: 1.531722
(Iteration 3471 / 4900) loss: 1.269929
(Iteration 3481 / 4900) loss: 1.389277
(Iteration 3491 / 4900) loss: 1.342569
(Iteration 3501 / 4900) loss: 1.532342
(Iteration 3511 / 4900) loss: 1.249602
(Iteration 3521 / 4900) loss: 1.286988
(Iteration 3531 / 4900) loss: 1.255366
(Iteration 3541 / 4900) loss: 1.320790
(Iteration 3551 / 4900) loss: 1.516495
(Iteration 3561 / 4900) loss: 1.215218
(Iteration 3571 / 4900) loss: 1.412340
(Iteration 3581 / 4900) loss: 1.514988
(Iteration 3591 / 4900) loss: 1.379671
(Iteration 3601 / 4900) loss: 1.527592
(Iteration 3611 / 4900) loss: 1.288664
(Iteration 3621 / 4900) loss: 1.349720
(Iteration 3631 / 4900) loss: 1.207210
(Iteration 3641 / 4900) loss: 1.457752
(Iteration 3651 / 4900) loss: 1.273430
(Iteration 3661 / 4900) loss: 1.369114
(Iteration 3671 / 4900) loss: 1.079390
(Iteration 3681 / 4900) loss: 1.162339
(Iteration 3691 / 4900) loss: 1.404993
(Iteration 3701 / 4900) loss: 1.254028
(Iteration 3711 / 4900) loss: 1.342848
(Iteration 3721 / 4900) loss: 1.412036
```

```
(Iteration 3731 / 4900) loss: 1.413683
(Iteration 3741 / 4900) loss: 1.153195
(Iteration 3751 / 4900) loss: 1.423030
(Iteration 3761 / 4900) loss: 1.224020
(Iteration 3771 / 4900) loss: 1.257108
(Iteration 3781 / 4900) loss: 1.246239
(Iteration 3791 / 4900) loss: 1.207203
(Iteration 3801 / 4900) loss: 1.351042
(Iteration 3811 / 4900) loss: 1.274750
(Iteration 3821 / 4900) loss: 1.344622
(Iteration 3831 / 4900) loss: 1.230457
(Iteration 3841 / 4900) loss: 1.293691
(Iteration 3851 / 4900) loss: 1.332515
(Iteration 3861 / 4900) loss: 1.278341
(Iteration 3871 / 4900) loss: 1.402976
(Iteration 3881 / 4900) loss: 1.201823
(Iteration 3891 / 4900) loss: 1.275469
(Iteration 3901 / 4900) loss: 1.286439
(Iteration 3911 / 4900) loss: 1.313085
(Epoch 8 / 10) train acc: 0.539000; val_acc: 0.470000
(Iteration 3921 / 4900) loss: 1.361607
(Iteration 3931 / 4900) loss: 1.055073
(Iteration 3941 / 4900) loss: 1.302249
(Iteration 3951 / 4900) loss: 1.214738
(Iteration 3961 / 4900) loss: 1.270087
(Iteration 3971 / 4900) loss: 1.189168
(Iteration 3981 / 4900) loss: 1.368281
(Iteration 3991 / 4900) loss: 1.269342
(Iteration 4001 / 4900) loss: 1.338180
(Iteration 4011 / 4900) loss: 1.582942
(Iteration 4021 / 4900) loss: 1.342648
(Iteration 4031 / 4900) loss: 1.476253
(Iteration 4041 / 4900) loss: 1.536958
(Iteration 4051 / 4900) loss: 1.091188
(Iteration 4061 / 4900) loss: 1.206614
(Iteration 4071 / 4900) loss: 1.333159
(Iteration 4081 / 4900) loss: 1.283151
(Iteration 4091 / 4900) loss: 1.233020
(Iteration 4101 / 4900) loss: 1.306298
(Iteration 4111 / 4900) loss: 1.354068
(Iteration 4121 / 4900) loss: 1.308154
(Iteration 4131 / 4900) loss: 1.302693
(Iteration 4141 / 4900) loss: 1.257422
(Iteration 4151 / 4900) loss: 1.474897
(Iteration 4161 / 4900) loss: 1.237975
(Iteration 4171 / 4900) loss: 1.333967
(Iteration 4181 / 4900) loss: 1.350034
(Iteration 4191 / 4900) loss: 1.500873
```

```
(Iteration 4201 / 4900) loss: 1.150138
(Iteration 4211 / 4900) loss: 1.290720
(Iteration 4221 / 4900) loss: 1.226863
(Iteration 4231 / 4900) loss: 1.233396
(Iteration 4241 / 4900) loss: 1.552031
(Iteration 4251 / 4900) loss: 1.242719
(Iteration 4261 / 4900) loss: 1.359631
(Iteration 4271 / 4900) loss: 1.502456
(Iteration 4281 / 4900) loss: 1.006978
(Iteration 4291 / 4900) loss: 1.390941
(Iteration 4301 / 4900) loss: 1.323812
(Iteration 4311 / 4900) loss: 1.274549
(Iteration 4321 / 4900) loss: 1.243123
(Iteration 4331 / 4900) loss: 1.419684
(Iteration 4341 / 4900) loss: 1.255258
(Iteration 4351 / 4900) loss: 1.244089
(Iteration 4361 / 4900) loss: 1.138581
(Iteration 4371 / 4900) loss: 1.283863
(Iteration 4381 / 4900) loss: 1.238588
(Iteration 4391 / 4900) loss: 1.312638
(Iteration 4401 / 4900) loss: 1.588226
(Epoch 9 / 10) train acc: 0.527000; val_acc: 0.456000
(Iteration 4411 / 4900) loss: 1.435487
(Iteration 4421 / 4900) loss: 1.154479
(Iteration 4431 / 4900) loss: 1.392360
(Iteration 4441 / 4900) loss: 1.245876
(Iteration 4451 / 4900) loss: 1.200481
(Iteration 4461 / 4900) loss: 1.366148
(Iteration 4471 / 4900) loss: 1.580206
(Iteration 4481 / 4900) loss: 1.398620
(Iteration 4491 / 4900) loss: 1.232539
(Iteration 4501 / 4900) loss: 1.422107
(Iteration 4511 / 4900) loss: 1.303118
(Iteration 4521 / 4900) loss: 1.165444
(Iteration 4531 / 4900) loss: 1.269013
(Iteration 4541 / 4900) loss: 1.209535
(Iteration 4551 / 4900) loss: 1.085849
(Iteration 4561 / 4900) loss: 1.383232
(Iteration 4571 / 4900) loss: 1.647506
(Iteration 4581 / 4900) loss: 1.482263
(Iteration 4591 / 4900) loss: 1.255463
(Iteration 4601 / 4900) loss: 1.228132
(Iteration 4611 / 4900) loss: 1.224333
(Iteration 4621 / 4900) loss: 1.423438
(Iteration 4631 / 4900) loss: 1.194008
(Iteration 4641 / 4900) loss: 1.098589
(Iteration 4651 / 4900) loss: 1.391406
(Iteration 4661 / 4900) loss: 1.287083
```

```
(Iteration 4671 / 4900) loss: 1.314095
(Iteration 4681 / 4900) loss: 1.432126
(Iteration 4691 / 4900) loss: 1.268540
(Iteration 4701 / 4900) loss: 1.093032
(Iteration 4711 / 4900) loss: 1.422660
(Iteration 4721 / 4900) loss: 1.379532
(Iteration 4731 / 4900) loss: 1.131978
(Iteration 4741 / 4900) loss: 1.463277
(Iteration 4751 / 4900) loss: 1.105897
(Iteration 4761 / 4900) loss: 1.435763
(Iteration 4771 / 4900) loss: 1.477966
(Iteration 4781 / 4900) loss: 1.378904
(Iteration 4791 / 4900) loss: 1.210157
(Iteration 4801 / 4900) loss: 1.233185
(Iteration 4811 / 4900) loss: 1.362364
(Iteration 4821 / 4900) loss: 1.410646
(Iteration 4831 / 4900) loss: 1.138897
(Iteration 4841 / 4900) loss: 1.470080
(Iteration 4851 / 4900) loss: 1.442729
(Iteration 4861 / 4900) loss: 1.183950
(Iteration 4871 / 4900) loss: 1.266656
(Iteration 4881 / 4900) loss: 1.344544
(Iteration 4891 / 4900) loss: 1.317356
(Epoch 10 / 10) train acc: 0.574000; val_acc: 0.482000
```

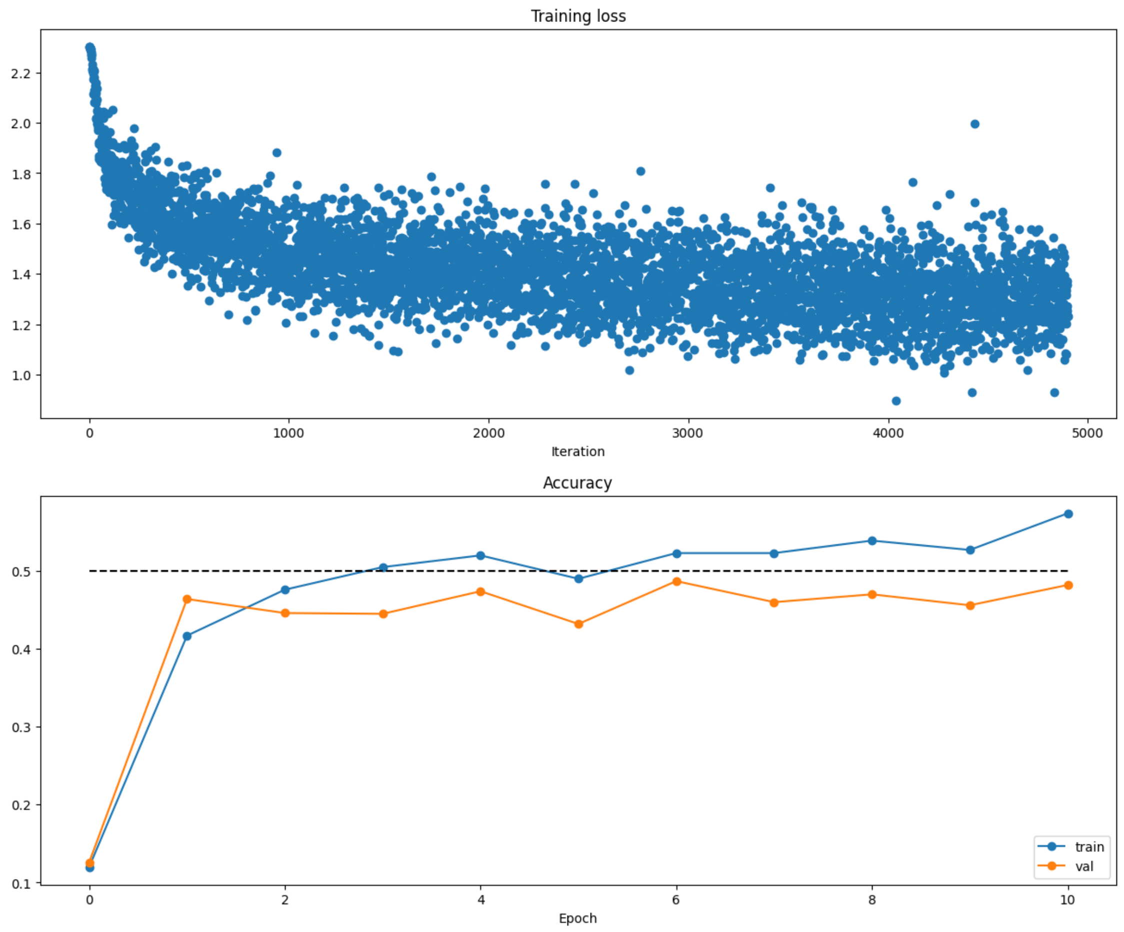
10 Debug the training

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.

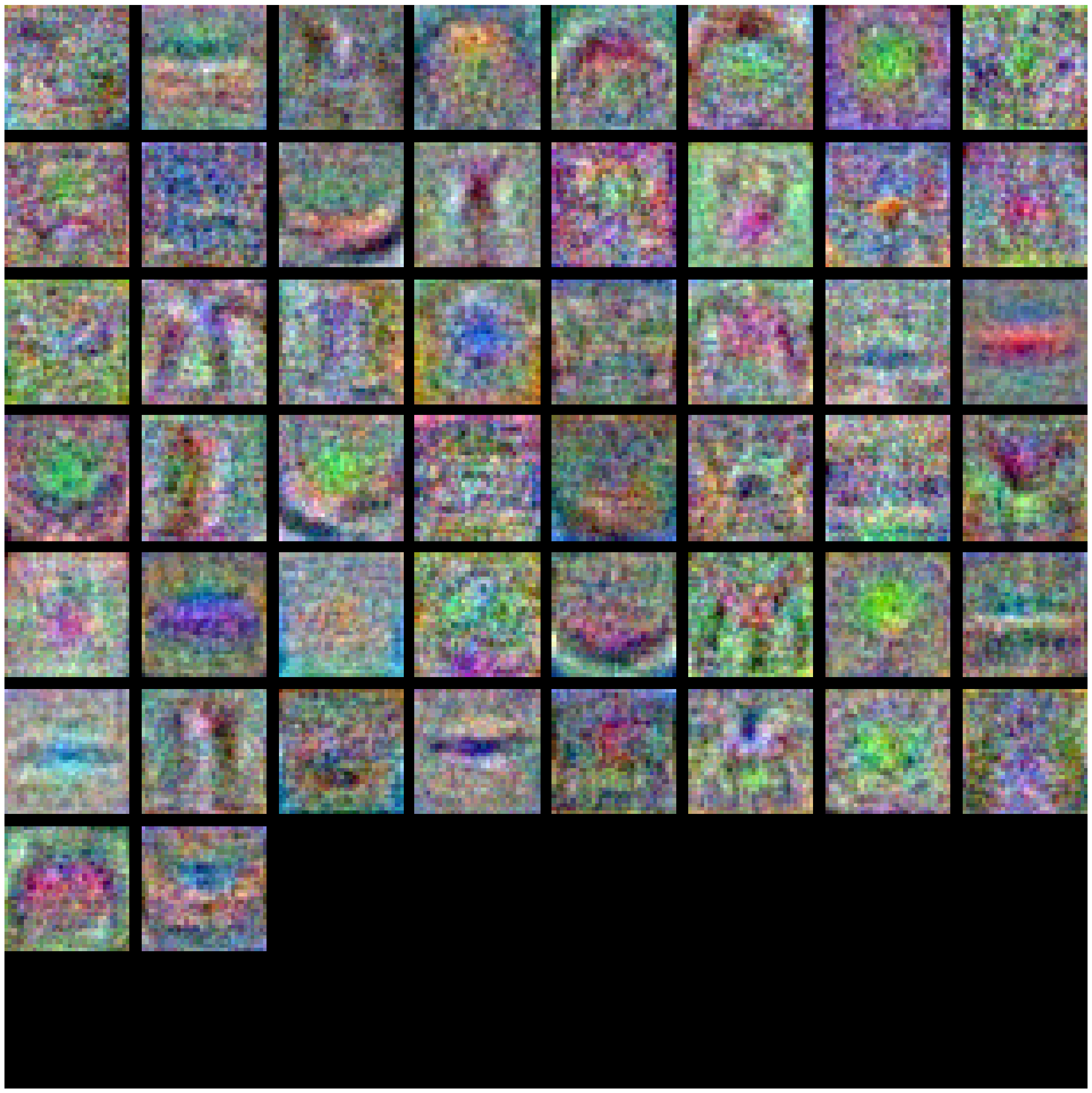
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[19]:
```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()
```



[20]:
```
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.).

[22]:
```
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)*****

best_val_acc = -1  # Initialize best validation accuracy

# Define hyperparameter ranges
learning_rates = [1e-3, 5e-4, 1e-4]
regularization_strengths = [0.1, 0.01, 0.001]
hidden_sizes = [50, 100, 200]
batch_sizes = [100, 200]
num_epochs = 10

# Hyperparameter tuning loop
```

```
for lr in learning_rates:
    for reg in regularization_strengths:
        for hidden_size in hidden_sizes:
            for batch_size in batch_sizes:

                # Initialize a TwoLayerNet with current hyperparameters
                model = TwoLayerNet(input_size, hidden_size, num_classes,␣
 ↪reg=reg)

                # Configure the solver
                solver = Solver(
                    model,
                    data,
                    update_rule='sgd',
                    optim_config={
                        'learning_rate': lr,
                    },
                    lr_decay=0.95,
                    num_epochs=num_epochs,
                    batch_size=batch_size,
                    verbose=False
                )

                # Train the model
                solver.train()

                # Evaluate validation accuracy
                val_acc = solver.check_accuracy(data['X_val'], data['y_val'])

                # If this is the best model, store it
                if val_acc > best_val_acc:
                    best_val_acc = val_acc
                    best_model = model

                print(f"lr: {lr}, reg: {reg}, hidden_size: {hidden_size},␣
 ↪batch_size: {batch_size}, val_acc: {val_acc}")

print(f"Best validation accuracy: {best_val_acc}")

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

```
lr: 0.001, reg: 0.1, hidden_size: 50, batch_size: 100, val_acc: 0.503
lr: 0.001, reg: 0.1, hidden_size: 50, batch_size: 200, val_acc: 0.507
lr: 0.001, reg: 0.1, hidden_size: 100, batch_size: 100, val_acc: 0.532
```

```
lr: 0.001, reg: 0.1, hidden_size: 100, batch_size: 200, val_acc: 0.523
lr: 0.001, reg: 0.1, hidden_size: 200, batch_size: 100, val_acc: 0.522
lr: 0.001, reg: 0.1, hidden_size: 200, batch_size: 200, val_acc: 0.53
lr: 0.001, reg: 0.01, hidden_size: 50, batch_size: 100, val_acc: 0.5
lr: 0.001, reg: 0.01, hidden_size: 50, batch_size: 200, val_acc: 0.511
lr: 0.001, reg: 0.01, hidden_size: 100, batch_size: 100, val_acc: 0.511
lr: 0.001, reg: 0.01, hidden_size: 100, batch_size: 200, val_acc: 0.505
lr: 0.001, reg: 0.01, hidden_size: 200, batch_size: 100, val_acc: 0.53
lr: 0.001, reg: 0.01, hidden_size: 200, batch_size: 200, val_acc: 0.525
lr: 0.001, reg: 0.001, hidden_size: 50, batch_size: 100, val_acc: 0.518
lr: 0.001, reg: 0.001, hidden_size: 50, batch_size: 200, val_acc: 0.506
lr: 0.001, reg: 0.001, hidden_size: 100, batch_size: 100, val_acc: 0.522
lr: 0.001, reg: 0.001, hidden_size: 100, batch_size: 200, val_acc: 0.524
lr: 0.001, reg: 0.001, hidden_size: 200, batch_size: 100, val_acc: 0.518
lr: 0.001, reg: 0.001, hidden_size: 200, batch_size: 200, val_acc: 0.526
lr: 0.0005, reg: 0.1, hidden_size: 50, batch_size: 100, val_acc: 0.515
lr: 0.0005, reg: 0.1, hidden_size: 50, batch_size: 200, val_acc: 0.497
lr: 0.0005, reg: 0.1, hidden_size: 100, batch_size: 100, val_acc: 0.514
lr: 0.0005, reg: 0.1, hidden_size: 100, batch_size: 200, val_acc: 0.513
lr: 0.0005, reg: 0.1, hidden_size: 200, batch_size: 100, val_acc: 0.53
lr: 0.0005, reg: 0.1, hidden_size: 200, batch_size: 200, val_acc: 0.518
lr: 0.0005, reg: 0.01, hidden_size: 50, batch_size: 100, val_acc: 0.512
lr: 0.0005, reg: 0.01, hidden_size: 50, batch_size: 200, val_acc: 0.502
lr: 0.0005, reg: 0.01, hidden_size: 100, batch_size: 100, val_acc: 0.516
lr: 0.0005, reg: 0.01, hidden_size: 100, batch_size: 200, val_acc: 0.506
lr: 0.0005, reg: 0.01, hidden_size: 200, batch_size: 100, val_acc: 0.534
lr: 0.0005, reg: 0.01, hidden_size: 200, batch_size: 200, val_acc: 0.512
lr: 0.0005, reg: 0.001, hidden_size: 50, batch_size: 100, val_acc: 0.52
lr: 0.0005, reg: 0.001, hidden_size: 50, batch_size: 200, val_acc: 0.504
lr: 0.0005, reg: 0.001, hidden_size: 100, batch_size: 100, val_acc: 0.517
lr: 0.0005, reg: 0.001, hidden_size: 100, batch_size: 200, val_acc: 0.505
lr: 0.0005, reg: 0.001, hidden_size: 200, batch_size: 100, val_acc: 0.527
lr: 0.0005, reg: 0.001, hidden_size: 200, batch_size: 200, val_acc: 0.523
lr: 0.0001, reg: 0.1, hidden_size: 50, batch_size: 100, val_acc: 0.465
lr: 0.0001, reg: 0.1, hidden_size: 50, batch_size: 200, val_acc: 0.421
lr: 0.0001, reg: 0.1, hidden_size: 100, batch_size: 100, val_acc: 0.468
lr: 0.0001, reg: 0.1, hidden_size: 100, batch_size: 200, val_acc: 0.427
lr: 0.0001, reg: 0.1, hidden_size: 200, batch_size: 100, val_acc: 0.479
lr: 0.0001, reg: 0.1, hidden_size: 200, batch_size: 200, val_acc: 0.444
lr: 0.0001, reg: 0.01, hidden_size: 50, batch_size: 100, val_acc: 0.467
lr: 0.0001, reg: 0.01, hidden_size: 50, batch_size: 200, val_acc: 0.426
lr: 0.0001, reg: 0.01, hidden_size: 100, batch_size: 100, val_acc: 0.478
lr: 0.0001, reg: 0.01, hidden_size: 100, batch_size: 200, val_acc: 0.434
lr: 0.0001, reg: 0.01, hidden_size: 200, batch_size: 100, val_acc: 0.48
lr: 0.0001, reg: 0.01, hidden_size: 200, batch_size: 200, val_acc: 0.449
lr: 0.0001, reg: 0.001, hidden_size: 50, batch_size: 100, val_acc: 0.463
lr: 0.0001, reg: 0.001, hidden_size: 50, batch_size: 200, val_acc: 0.414
lr: 0.0001, reg: 0.001, hidden_size: 100, batch_size: 100, val_acc: 0.465
```

```
lr: 0.0001, reg: 0.001, hidden_size: 100, batch_size: 200, val_acc: 0.438
lr: 0.0001, reg: 0.001, hidden_size: 200, batch_size: 100, val_acc: 0.483
lr: 0.0001, reg: 0.001, hidden_size: 200, batch_size: 200, val_acc: 0.451
Best validation accuracy: 0.534
```

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.

[23]:
```
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:  0.534
```

[24]:
```
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:  0.522
```

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. Train on a larger dataset. 2. Increase the regularization strength.

YourExplanation : 1. While training on a larger dateset will decrease the gap because it often reduces overfitting since the model is more exposed to more diverse examples.

2. Increasing the regularization strength will penalize overly complex model and prevent it to fit the noise in the traning data.

[ ]:
```

```

features

September 11, 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.

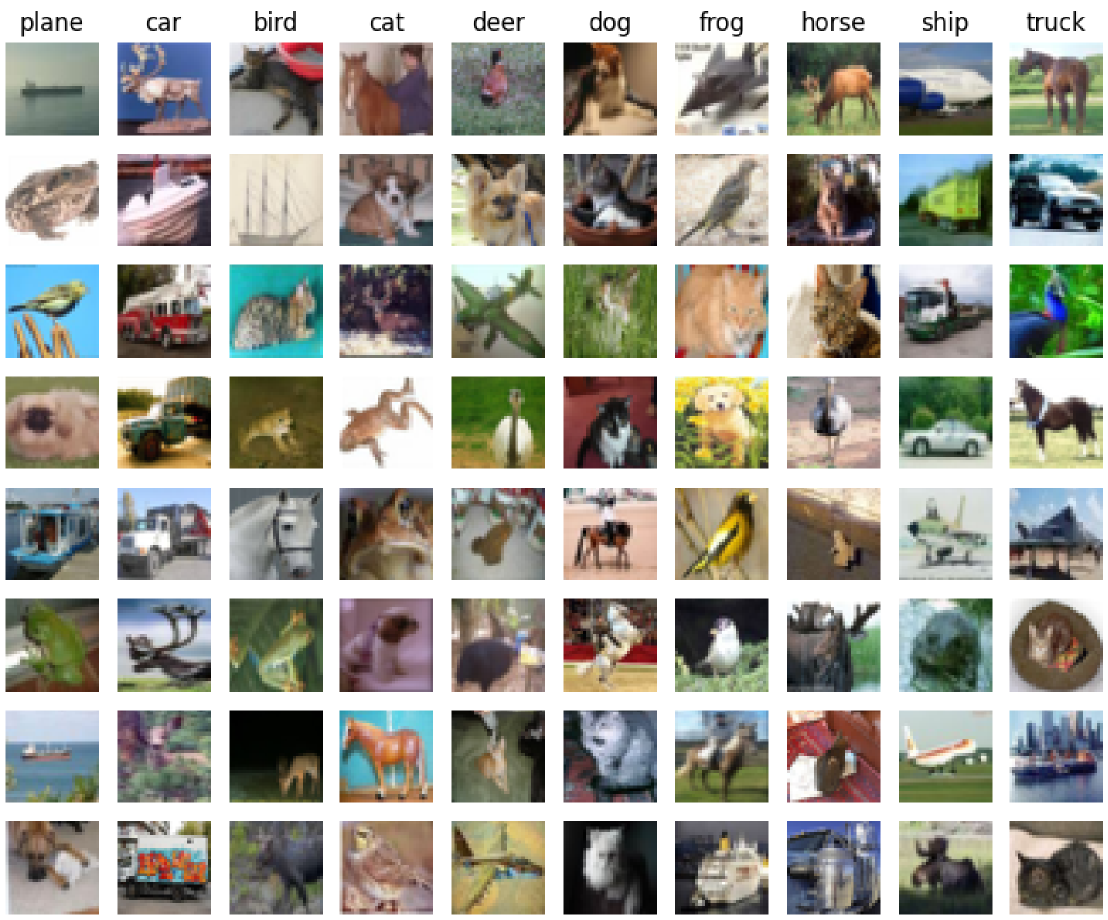
[9]:
```
# Use the validation set to tune the learning rate and regularization strength

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 reg in regularization_strengths:
        svm = LinearSVM()

        # Train on extracted features (instead of raw image data)
        svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,␣
 ↪num_iters=1500, verbose=False)

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

        # Calculate accuracy
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # Track the best model
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm


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

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

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

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.107898 val accuracy: 0.121000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.080837 val accuracy: 0.099000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414898 val accuracy: 0.420000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.138694 val accuracy: 0.136000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416816 val accuracy: 0.423000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.405082 val accuracy: 0.413000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415857 val accuracy: 0.414000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.400265 val accuracy: 0.397000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.332673 val accuracy: 0.320000
best validation accuracy achieved: 0.423000
```

[10]:
```
# 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
```

[11]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

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



1.3.1 Inline question 1:

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

Your Answer :

Some images have blurry or unclear features, leading to the possibility of misclassification by the model due to lack of clear distinctive features.

Yes, the misclassifications seem reasonable because, in many cases, the visual features of certain objects or animals overlap.

For example:

Frogs and other small animals (birds or reptiles) could get mixed up due to their small size and often similar environments.

Cars and trucks can look alike depending on the angle or the shape of the vehicle, making it harder to distinguish them.

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.

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

[13]:
```
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
```

```python
################################################################################
# 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)*****

best_val_acc = -1

# Hyperparameter tuning variables
learning_rates = [1e-3, 1e-2, 1e-1]
regularization_strengths = [0.1, 0.01, 0.001]

best_val_acc = -1
best_net = None

# Loop over hyperparameters
for lr in learning_rates:
    for reg in regularization_strengths:
        # Initialize the two-layer network
        net = TwoLayerNet(input_dim, hidden_dim, num_classes, reg=reg)

        # Set up the solver
        solver = Solver(net, data,
                        update_rule='sgd',
                        optim_config={'learning_rate': lr},
                        lr_decay=0.95,
                        num_epochs=10,
                        batch_size=200,
                        verbose=False)

        # Train the model
        solver.train()

        # Check the validation accuracy
        val_acc = solver.check_accuracy(X_val_feats, y_val)

        # Keep track of the best model based on validation accuracy
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_net = net

        print(f'lr: {lr}, reg: {reg}, val_acc: {val_acc}')

# Print the best validation accuracy achieved
print(f'Best validation accuracy: {best_val_acc}')
```

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

```
lr: 0.001, reg: 0.1, val_acc: 0.167
lr: 0.001, reg: 0.01, val_acc: 0.206
lr: 0.001, reg: 0.001, val_acc: 0.185
lr: 0.01, reg: 0.1, val_acc: 0.265
lr: 0.01, reg: 0.01, val_acc: 0.343
lr: 0.01, reg: 0.001, val_acc: 0.356
lr: 0.1, reg: 0.1, val_acc: 0.444
lr: 0.1, reg: 0.01, val_acc: 0.534
lr: 0.1, reg: 0.001, val_acc: 0.558
Best validation accuracy: 0.558
```

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

```
0.551
```