knn

October 14, 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. 'cse493g1/assignments/assignment1/'
     FOLDERNAME = 'cse493g1/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/$FOLDERNAME/cse493g1/datasets/
     !bash get_datasets.sh
     %cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/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.

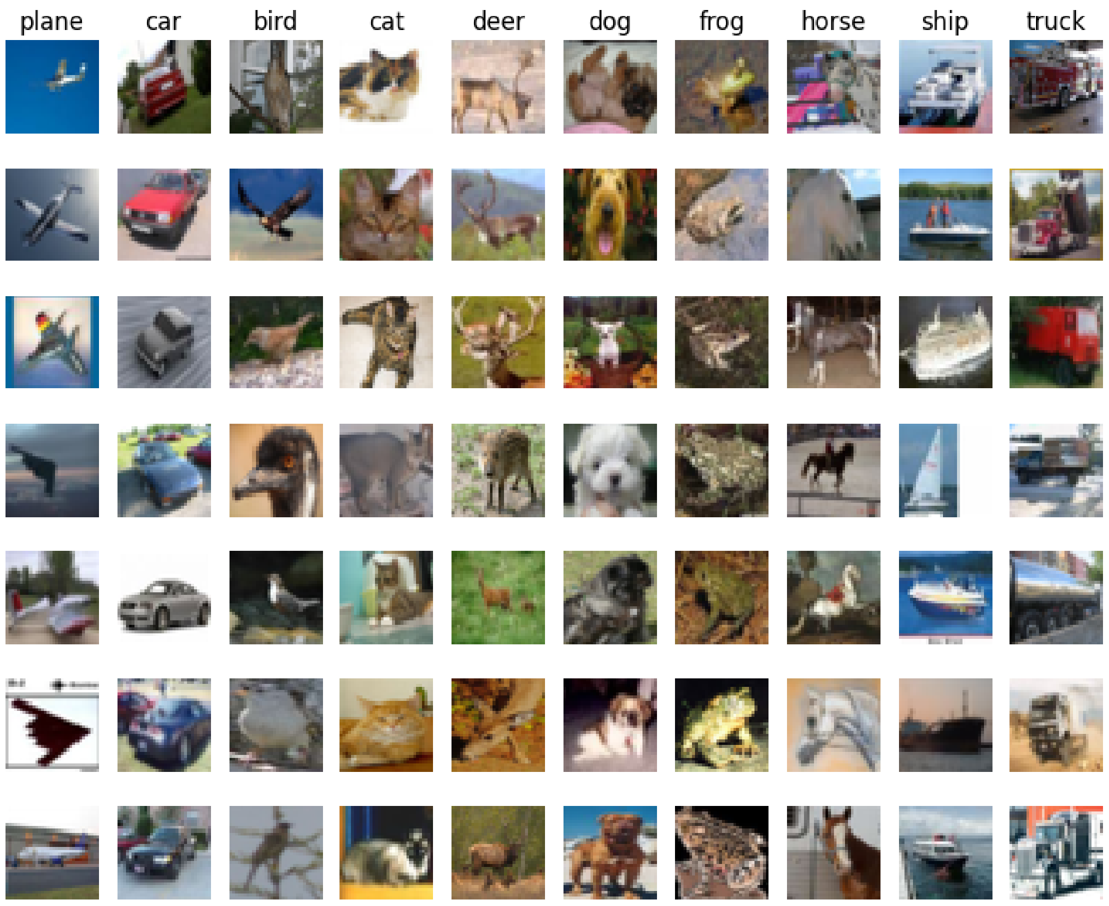
[ ]:
```python
# Run some setup code for this notebook.
import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

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

[ ]:
```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'cse493g1/datasets/cifar-10-batches-py'

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

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

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

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

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



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

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

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

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

[ ]:
```
from cse493g1.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.

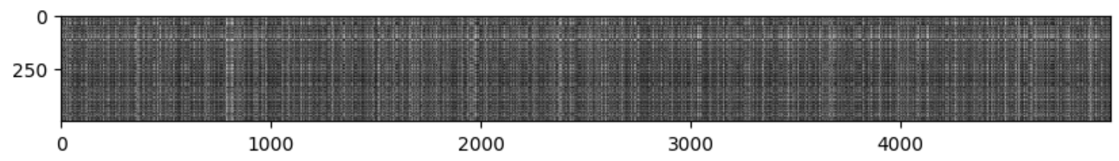
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cse493g1/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 cse493g1/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?

$Your Answer:$

For the data that causes the distinctly bright rows it would be TEST set images that has a L_2 distance thats quite high. In more general terms the bright rows are TEST images that have very different individual pixel values from the TRAIN images they are being compared to. \ Then for the brigth columns that would mean that certain TRAIN set imgaes have the same characterisitc where their individual pixel values are quite different from the TEST set imgaes they are being compared to.

[ ]:
```
# 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=4)

# 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 136 / 500 correct => accuracy: 0.272000
```

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. 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.

$\mathit{Your Answer:}$

1, 2, 3

$\mathit{Your Explanation:}$

1. No Difference: subtracting by a constant changes the distance between all images the same amount.

2. No Differnence: Similar to 1 where subtracting by a constant doesn't change the distance between two values here each pixel is shifted the same amount so the distance between two pixels is preserved.

3. No difference: Subtracting by a constant doesn't change the distance, and since all the images are being scaled by a constant amount in every direction the distance is still preserved. This is also the normalization process where after these two opperations the image pixel values have $\mu = 0$ and $\sigma = 1$ for the set of images.

4. There is a difference: Subtracting by the pixel-wise mean doesn't do anything, but our problem is the pixel-wise standard deviation. This is because suppose we think of each of our pixels as a dimension or data point if we multiply each dimension by a diffferent scalar then some dimensions will have more or less weight. But for L1 distance we can't say that a certain dimension or axis is more imporant.

5. there may or may not be a difference:

If we are rotating the entire coodinate axes of our classifier then the distance would be changed since L1 distance is not rotationally invariant. So thinking of our image as a multi-dimensional vector if we roate the vector in some sense then $L1$ distance can change drastically.

When it won't have a difference is when all the images are rotated the same amount with enough padding and there is no pixel-loss or cuts. This would be where all the images are rotated 90° since then no pixel is cut off and image to image (pixel-pixel) the distance would stay the same. So the nearest neighbor is still the same

[ ]:
```
# 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 41.369972 seconds
One loop version took 61.575044 seconds
No loop version took 1.080956 seconds
```

1.0.1 Cross-validation

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

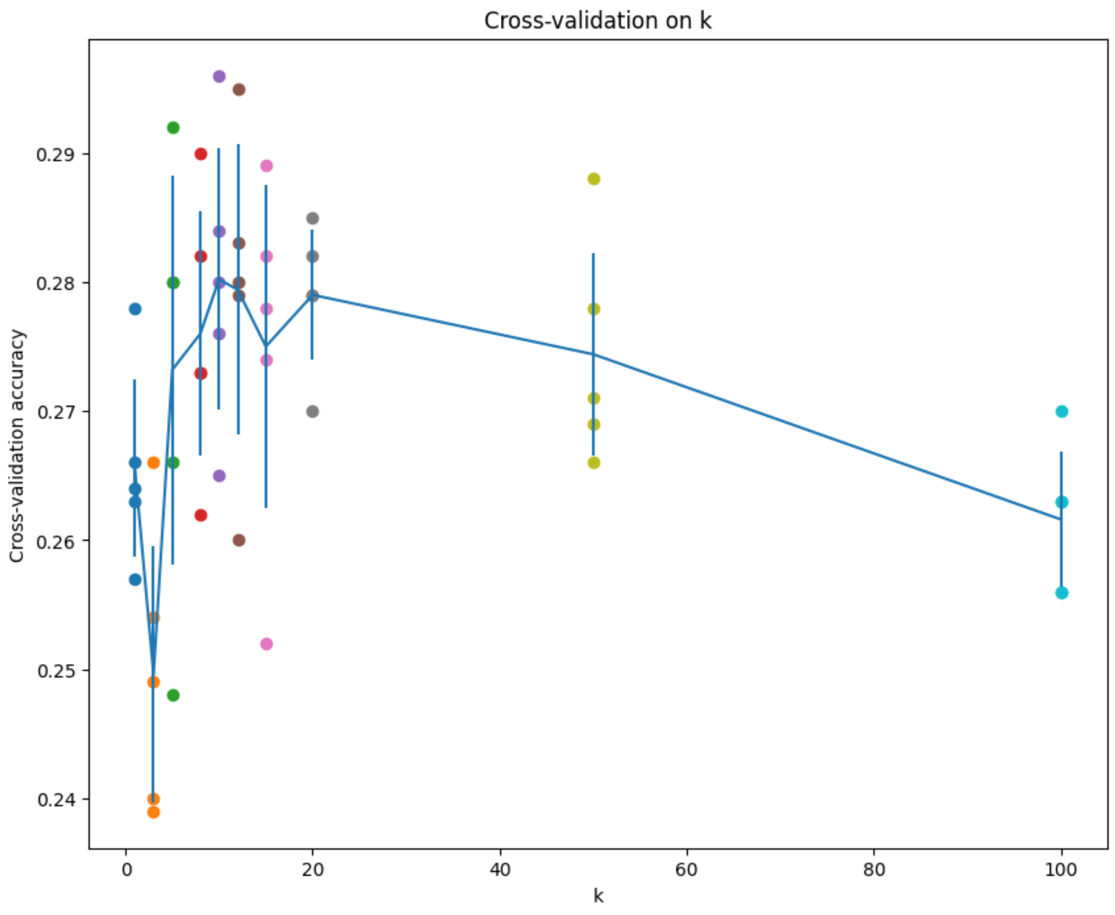
[ ]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
```

```
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
pass

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

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


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

for k in k_choices:
  k_to_accuracies[k] = []
  for i in range(num_folds):
    valset = X_train_folds[i]
    valsetLabels = y_train_folds[i]
    trainSet = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
    trainSetLabels = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])
    classifier.train(trainSet, trainSetLabels)
    dists = classifier.compute_distances_no_loops(valset)
    y_test_pred = classifier.predict_labels(dists, k)
    num_correct = np.sum(y_test_pred == valsetLabels)
    accuracy = float(num_correct) / valset.shape[0]
    k_to_accuracies[k].append(accuracy)
pass

# *****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 = 10

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

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

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

Inline Question 3

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

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

4

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

1. False - For example suppose that we have a 1-NN trying to classify an image as dog or cat. Then lets say all but 1 image is a dog in our train set and visualized on a 2d graph is right at $(0, 0)$ and all points inside a circle of radius 5 centered at $(0, 0)$ the nearest neighbor is the dog. That would mean that our decision boundry would be non-linear (effectively a circle) since all points where the dog picture isn't the nearest neighbhor would be classified as a cat.

2. False - Suppose the same example as before where we are trying to classify between dogs and cats there is only 1 dog picture then for $k = 1$ the classifier will have no training error. But for $k = 5$ even the training dog picture will be classified as the 4/5 of the nearest neighbors would be cats.

3. False - We just found above for the CIFAR-10 dataset that $k = 10$ was better than $k = 1$.

4. True - In our distance calculation after we set our Train points we take the distance from each Train point from our Test point then take the k closest points then take the majority labels from the k closet points. So we won't know which Train points are the closest to our Test point unless we get the distance from every Train point. So more Train points then more distances we have to calculate from each Test point.

5. False - #4 is true

svm

October 14, 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. 'cse493g1/assignments/assignment1/'
FOLDERNAME = 'cse493g1/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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/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

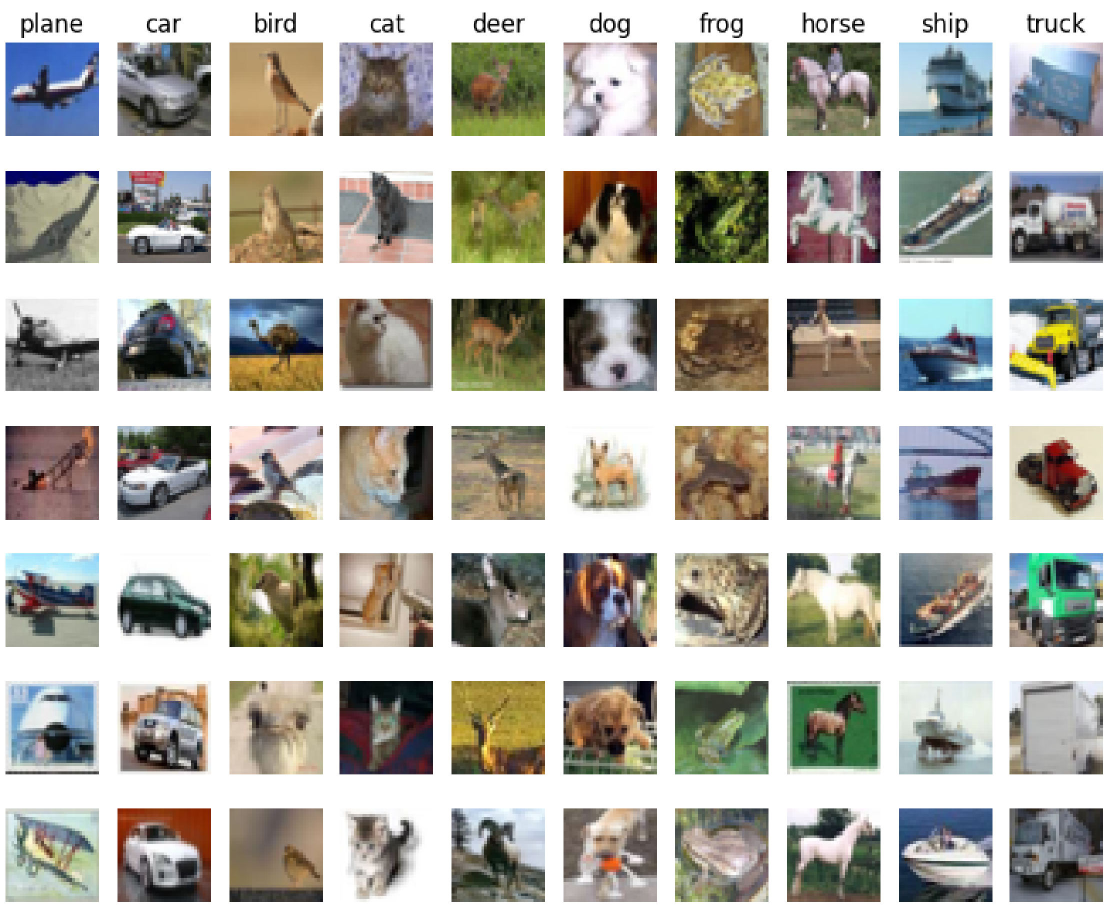
[ ]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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 = 'cse493g1/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)
```

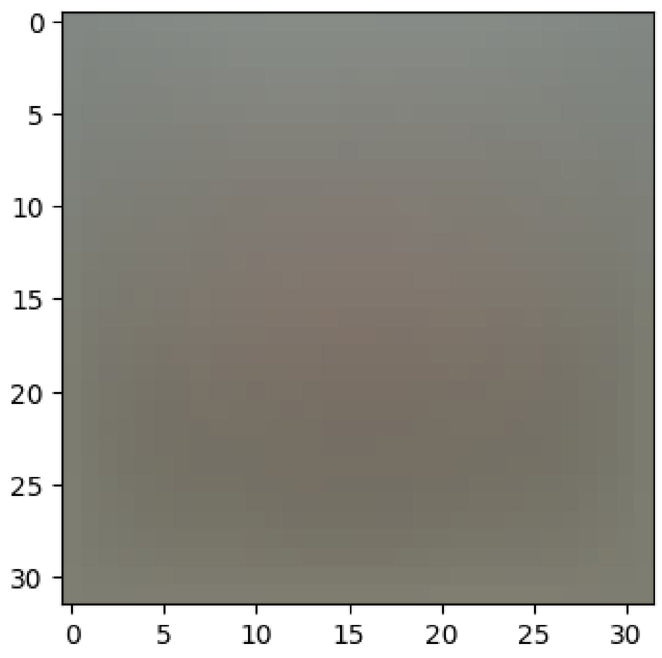
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

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

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

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

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

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

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

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



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

1.2 SVM Classifier

Your code for this section will all be written inside `cse493g1/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 cse493g1.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: 9.337673
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

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

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

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should␣
 ↪match
# almost exactly along all dimensions.
from cse493g1.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: 10.876248 analytic: 10.876248, relative error: 4.252829e-11
numerical: 12.825098 analytic: 12.825098, relative error: 1.608127e-12
numerical: 27.171252 analytic: 27.171252, relative error: 4.695459e-12
numerical: -19.995993 analytic: -19.937829, relative error: 1.456507e-03
numerical: -17.868780 analytic: -17.815730, relative error: 1.486642e-03
numerical: 13.520592 analytic: 13.520592, relative error: 2.815343e-11
numerical: -5.321421 analytic: -5.321421, relative error: 2.954224e-11
numerical: -3.127459 analytic: -3.127459, relative error: 4.679122e-11
numerical: -60.105772 analytic: -60.203657, relative error: 8.136130e-04
numerical: 4.310606 analytic: 4.310606, relative error: 7.259642e-11
numerical: 6.005082 analytic: 6.005082, relative error: 3.041278e-11
numerical: 18.113150 analytic: 18.113150, relative error: 5.642948e-13
numerical: -16.340198 analytic: -16.340198, relative error: 1.607344e-11
numerical: 27.003068 analytic: 27.003068, relative error: 1.284949e-11
numerical: -2.069577 analytic: -2.069577, relative error: 1.326697e-10
numerical: -2.423864 analytic: -2.423864, relative error: 1.151909e-10
numerical: 23.890664 analytic: 23.890664, relative error: 1.984490e-11
numerical: -14.869898 analytic: -14.856009, relative error: 4.672460e-04
numerical: -1.149531 analytic: -1.149531, relative error: 3.421576e-11
numerical: -19.276266 analytic: -19.237886, relative error: 9.965217e-04
```

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*

$Your Answer:$

- Yes there could be a missmatch.
- Time when gradcheck wouldn't match exactly could be due to multiple things, one would be h in the limit definition of a dertivate could cause a discrepency where the analytical derivative would be as exact as the computer would allow while the numerical derivative would be related to how small h is set to.

This could also be because of the fact that SVM is is not truly differentialable where for max(x,y) when $x = y$ you get a kink where the function is not differentiable but you can take the analytical derivative due to h. * It could be a reason for concern esspecially during when the analytical and numerical derivative don't match near the kink.

- Suppose that we have the function max(0,x) and $x = -2 * 10^{-10}$ and we have $h = 10^{-9}$ then for our numeric or computer derivative we get $\frac{(-2 \cdot 10^{-10} + 10^{-9}) + 2 \cdot 10^{-10}}{10^{-9}}$ which is 1. But our analytical deritive would be 0 so we now have a step value for when we do gradient decent that doesn't do what we want.
- Increasing the margin would decrease the change of landing on the kink since the postive increase in margin would push x into a more postive direction away from the point of the kink when all other variables are the same.

[ ]:
```
# 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 cse493g1.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: 9.337673e+00 computed in 0.102727s
Vectorized loss: 9.337673e+00 computed in 0.012634s
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 0.097651s
Vectorized loss and gradient: computed in 0.009043s
difference: 0.000000
```

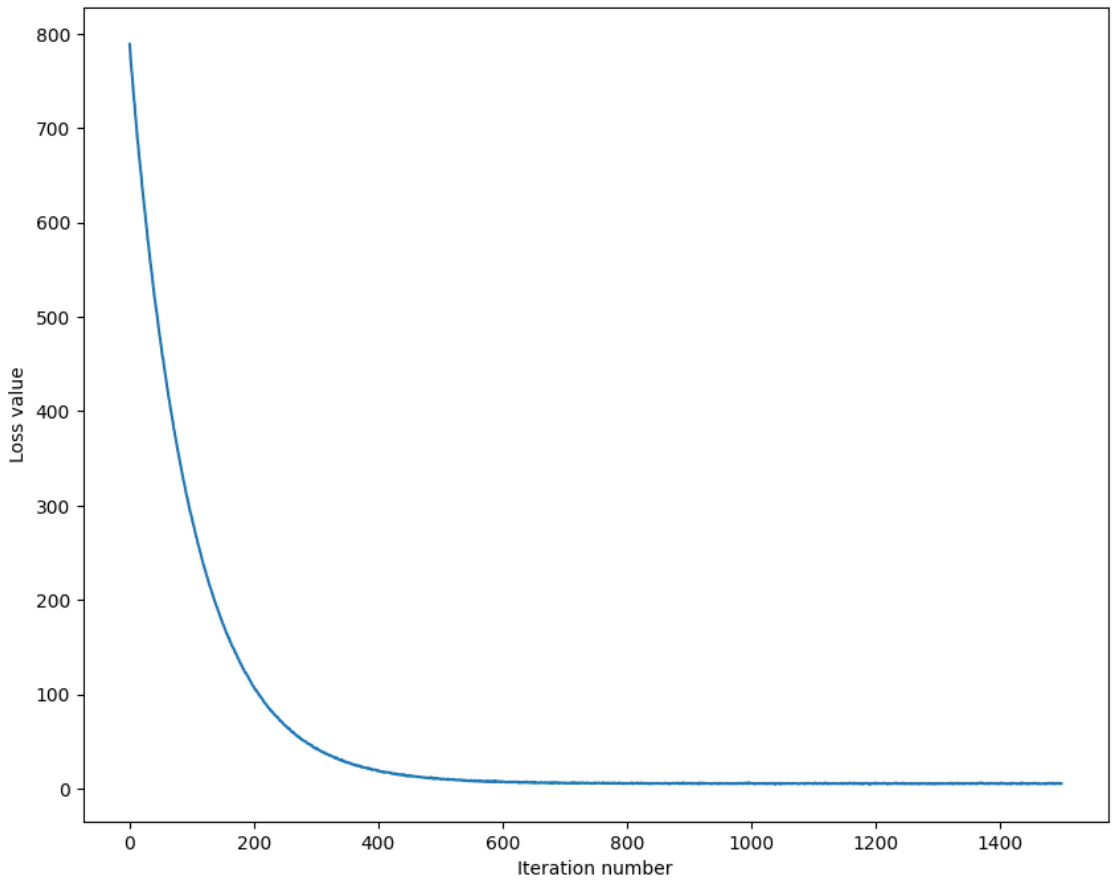
1.2.1 Stochastic Gradient Descent

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 `cse493g1/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 cse493g1.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 789.140074
iteration 100 / 1500: loss 287.138830
iteration 200 / 1500: loss 107.860482
iteration 300 / 1500: loss 43.196253
iteration 400 / 1500: loss 18.628786
iteration 500 / 1500: loss 10.128548
iteration 600 / 1500: loss 7.104128
iteration 700 / 1500: loss 5.679763
iteration 800 / 1500: loss 5.685376
iteration 900 / 1500: loss 5.665010
iteration 1000 / 1500: loss 5.552857
iteration 1100 / 1500: loss 5.511509
iteration 1200 / 1500: loss 6.171029
iteration 1300 / 1500: loss 5.395297
iteration 1400 / 1500: loss 5.621639
That took 6.875000s
```

[ ]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



[ ]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

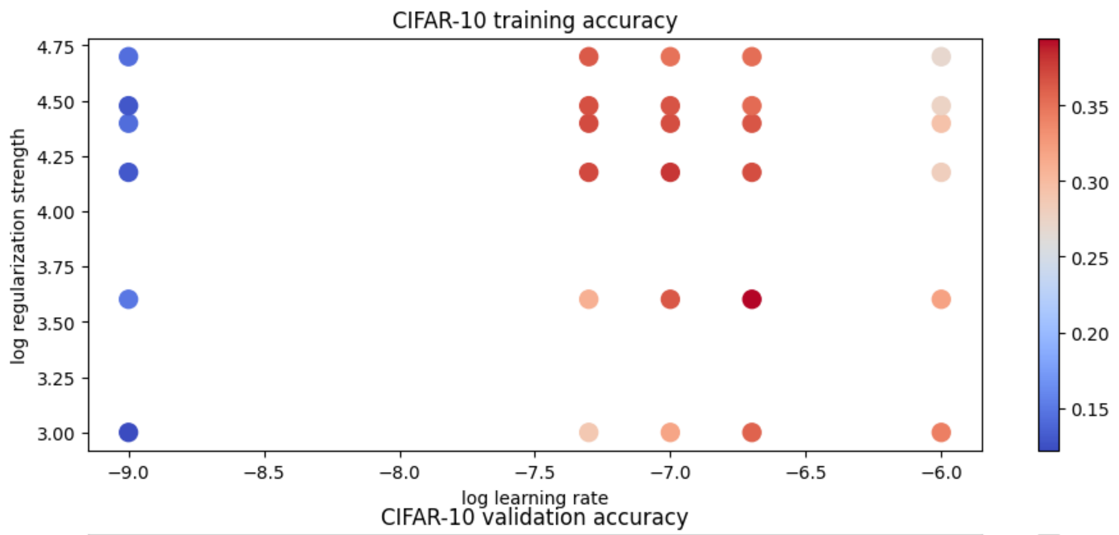
```
training accuracy: 0.366633
validation accuracy: 0.377000
```

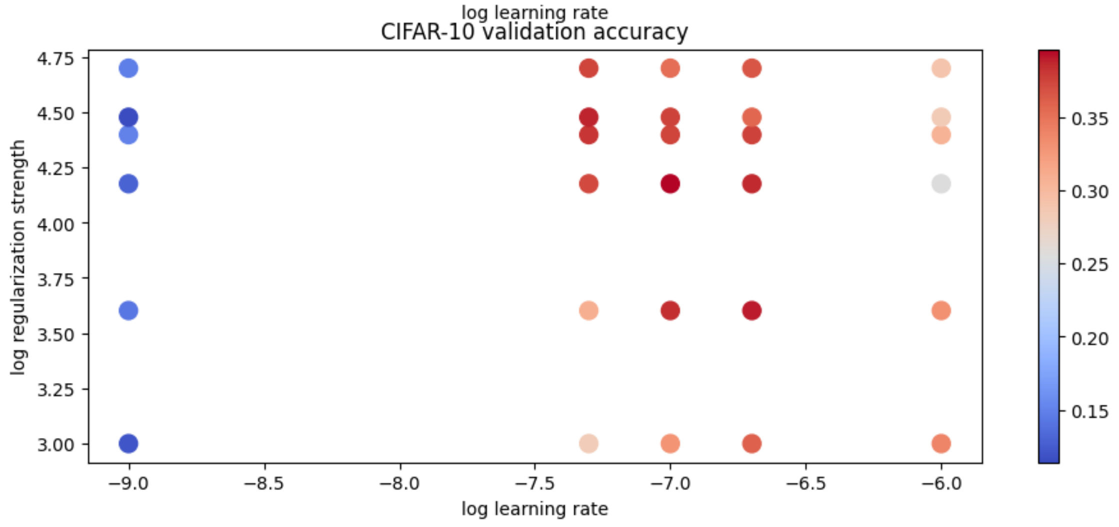
```
[ ]: # Use the validation set to tune hyperparameters (regularization strength and
     # learning rate). You should experiment with different ranges for the learning
     # rates and regularization strengths; if you are careful you should be able to
     # get a classification accuracy of about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

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

     # results is dictionary mapping tuples of the form
     # (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, 1e-9, 1e-6, 2e-7, 5e-8]
     regularization_strengths = [2.5e4, 5e4, 1e3, 3e4, 4e3, 1.5e4]

     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
     for i in learning_rates:
       for j in regularization_strengths:
         svm = LinearSVM()
         loss_hist = svm.train(X_train, y_train, learning_rate=i, reg=j,
                               num_iters=1500)
         y_train_pred = svm.predict(X_train)
         train_accuracy = np.mean(y_train == y_train_pred)
         y_val_pred = svm.predict(X_val)
         val_accuracy = np.mean(y_val == y_val_pred)
```

```
    results[(i,j)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm

pass

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

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

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

```
lr 1.000000e-09 reg 1.000000e+03 train accuracy: 0.121408 val accuracy: 0.120000
lr 1.000000e-09 reg 4.000000e+03 train accuracy: 0.148224 val accuracy: 0.142000
lr 1.000000e-09 reg 1.500000e+04 train accuracy: 0.129531 val accuracy: 0.129000
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.142571 val accuracy: 0.150000
lr 1.000000e-09 reg 3.000000e+04 train accuracy: 0.128204 val accuracy: 0.114000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.143816 val accuracy: 0.148000
lr 5.000000e-08 reg 1.000000e+03 train accuracy: 0.288449 val accuracy: 0.282000
lr 5.000000e-08 reg 4.000000e+03 train accuracy: 0.309653 val accuracy: 0.309000
lr 5.000000e-08 reg 1.500000e+04 train accuracy: 0.372184 val accuracy: 0.372000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.370612 val accuracy: 0.381000
lr 5.000000e-08 reg 3.000000e+04 train accuracy: 0.368653 val accuracy: 0.388000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.362633 val accuracy: 0.374000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.317857 val accuracy: 0.329000
lr 1.000000e-07 reg 4.000000e+03 train accuracy: 0.364714 val accuracy: 0.384000
lr 1.000000e-07 reg 1.500000e+04 train accuracy: 0.379837 val accuracy: 0.396000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.369367 val accuracy: 0.375000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.366653 val accuracy: 0.376000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.350592 val accuracy: 0.353000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.359163 val accuracy: 0.360000
lr 2.000000e-07 reg 4.000000e+03 train accuracy: 0.394633 val accuracy: 0.390000
lr 2.000000e-07 reg 1.500000e+04 train accuracy: 0.370020 val accuracy: 0.385000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.365673 val accuracy: 0.376000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.355143 val accuracy: 0.357000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.352286 val accuracy: 0.366000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.342408 val accuracy: 0.339000
lr 1.000000e-06 reg 4.000000e+03 train accuracy: 0.320204 val accuracy: 0.332000
lr 1.000000e-06 reg 1.500000e+04 train accuracy: 0.280918 val accuracy: 0.256000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.293776 val accuracy: 0.306000
lr 1.000000e-06 reg 3.000000e+04 train accuracy: 0.275469 val accuracy: 0.283000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.269102 val accuracy: 0.291000
best validation accuracy achieved during cross-validation: 0.396000
```

[24]:
```
# 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()
```



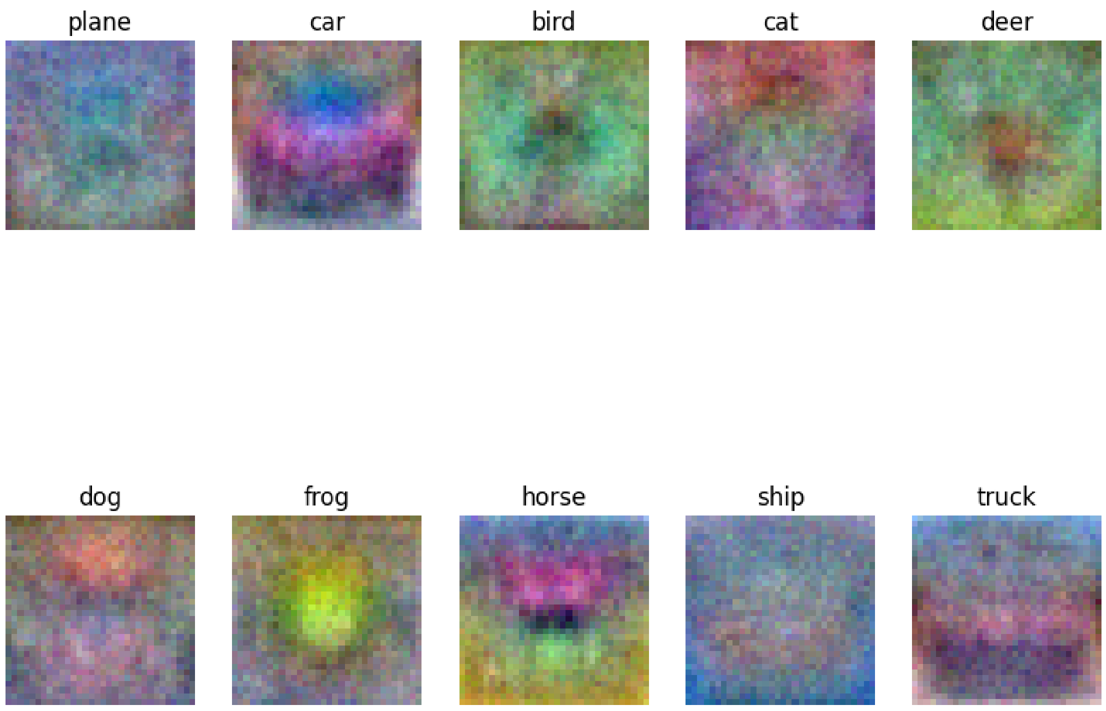


[25]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.382000
```

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

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

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



Inline question 2

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

$Your Answer:$

The SVM weights seem to look like templates for what the average picture in each class looks like. Where as the classifier optimizes for each class it converges towards a template or example picture that close to all the images in the class. So for example we can assume that most car images are ones taken facing the front of the car and as such the resulting weights for the class depict a car from the fronts. The same idea applies to the colors of the "templates" (SVM weights)

softmax

October 14, 2024

[2]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cse493g1/assignments/assignment1/'
FOLDERNAME = 'cse493g1/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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignment1
```

1 Softmax exercise

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

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

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

- **visualize** the final learned weights

[3]:
```
import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

[4]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cse493g1/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 `cse493g1/classifiers/softmax.py`.

[22]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cse493g1/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cse493g1.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: 2.352632
sanity check: 2.302585
```

Inline Question 1

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

$\color{blue}{\textit{Your Answer:}}$ This is because we can assume most of the initial weights will cause all scores to be equal so

$-log(1/c) = -log(1/10) = -log(0.1)$

and thats why most randomly intilaized W will give us a loss close to $-log(0.1)$

[23]:
```
# 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 cse493g1.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

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

```
numerical: 1.979549 analytic: 1.979549, relative error: 1.113721e-09
numerical: 0.079131 analytic: 0.079131, relative error: 2.138496e-07
numerical: -1.042820 analytic: -1.042820, relative error: 1.698593e-08
numerical: 3.796590 analytic: 3.796590, relative error: 1.201448e-08
numerical: 0.859505 analytic: 0.859505, relative error: 2.804714e-08
numerical: -0.576089 analytic: -0.576089, relative error: 2.486722e-08
numerical: -0.084776 analytic: -0.084776, relative error: 5.740196e-07
numerical: -0.420156 analytic: -0.420156, relative error: 9.969092e-08
numerical: -0.834440 analytic: -0.834440, relative error: 9.737449e-08
numerical: -0.008158 analytic: -0.008158, relative error: 4.043350e-06
numerical: -2.869425 analytic: -2.869425, relative error: 3.504249e-10
numerical: 0.229256 analytic: 0.229256, relative error: 1.493819e-07
numerical: 0.093495 analytic: 0.093495, relative error: 4.762652e-07
numerical: -0.059050 analytic: -0.059050, relative error: 1.825387e-07
numerical: 0.178142 analytic: 0.178142, relative error: 5.788260e-08
numerical: 2.707534 analytic: 2.707534, relative error: 4.773242e-09
numerical: 2.692399 analytic: 2.692399, relative error: 2.714903e-08
numerical: 0.282282 analytic: 0.282282, relative error: 7.453515e-09
numerical: -1.805005 analytic: -1.805005, relative error: 4.673921e-09
numerical: -1.288543 analytic: -1.288544, relative error: 3.807628e-08
```

[48]:

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

from cse493g1.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: 2.352632e+00 computed in 0.129094s
vectorized loss: 2.352632e+00 computed in 0.013047s
```

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

[55]:
```
# 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 cse493g1.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, 1e-9, 1e-6, 2e-7, 5e-8]
regularization_strengths = [2.5e4, 5e4, 1e3, 3e4, 4e3, 1.5e4]

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

for i in learning_rates:
  for j in regularization_strengths:
    softmax = Softmax()
    loss_hist = softmax.train(X_train, y_train, learning_rate=i, reg=j,
                          num_iters=1500)
    y_train_pred = softmax.predict(X_train)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = softmax.predict(X_val)
    val_accuracy = np.mean(y_val == y_val_pred)
    results[(i,j)] = (train_accuracy, val_accuracy)
    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-09 reg 1.000000e+03 train accuracy: 0.109122 val accuracy: 0.121000
lr 1.000000e-09 reg 4.000000e+03 train accuracy: 0.103163 val accuracy: 0.102000
lr 1.000000e-09 reg 1.500000e+04 train accuracy: 0.106000 val accuracy: 0.099000
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.106735 val accuracy: 0.117000
lr 1.000000e-09 reg 3.000000e+04 train accuracy: 0.101735 val accuracy: 0.115000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.089082 val accuracy: 0.082000
lr 5.000000e-08 reg 1.000000e+03 train accuracy: 0.225918 val accuracy: 0.229000
lr 5.000000e-08 reg 4.000000e+03 train accuracy: 0.247020 val accuracy: 0.278000
lr 5.000000e-08 reg 1.500000e+04 train accuracy: 0.325653 val accuracy: 0.341000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.323898 val accuracy: 0.337000
lr 5.000000e-08 reg 3.000000e+04 train accuracy: 0.320714 val accuracy: 0.328000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.306082 val accuracy: 0.329000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.261633 val accuracy: 0.274000
lr 1.000000e-07 reg 4.000000e+03 train accuracy: 0.317837 val accuracy: 0.314000
lr 1.000000e-07 reg 1.500000e+04 train accuracy: 0.337286 val accuracy: 0.359000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.325714 val accuracy: 0.348000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.325286 val accuracy: 0.343000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.310510 val accuracy: 0.326000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.315857 val accuracy: 0.337000
lr 2.000000e-07 reg 4.000000e+03 train accuracy: 0.375429 val accuracy: 0.382000
lr 2.000000e-07 reg 1.500000e+04 train accuracy: 0.348265 val accuracy: 0.356000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.325612 val accuracy: 0.343000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.319429 val accuracy: 0.341000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.304694 val accuracy: 0.315000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.396204 val accuracy: 0.394000
lr 1.000000e-06 reg 4.000000e+03 train accuracy: 0.372694 val accuracy: 0.384000
lr 1.000000e-06 reg 1.500000e+04 train accuracy: 0.338735 val accuracy: 0.349000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.330776 val accuracy: 0.337000
lr 1.000000e-06 reg 3.000000e+04 train accuracy: 0.313143 val accuracy: 0.329000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.289510 val accuracy: 0.292000
best validation accuracy achieved during cross-validation: 0.394000
```

[56]:
```
# 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: 0.395000
```

Inline Question 2 - *True or False*

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

$Your Answer:$

Yes, it is possible to add a data point that would leave SVM loss unchanged but not Softmax loss.

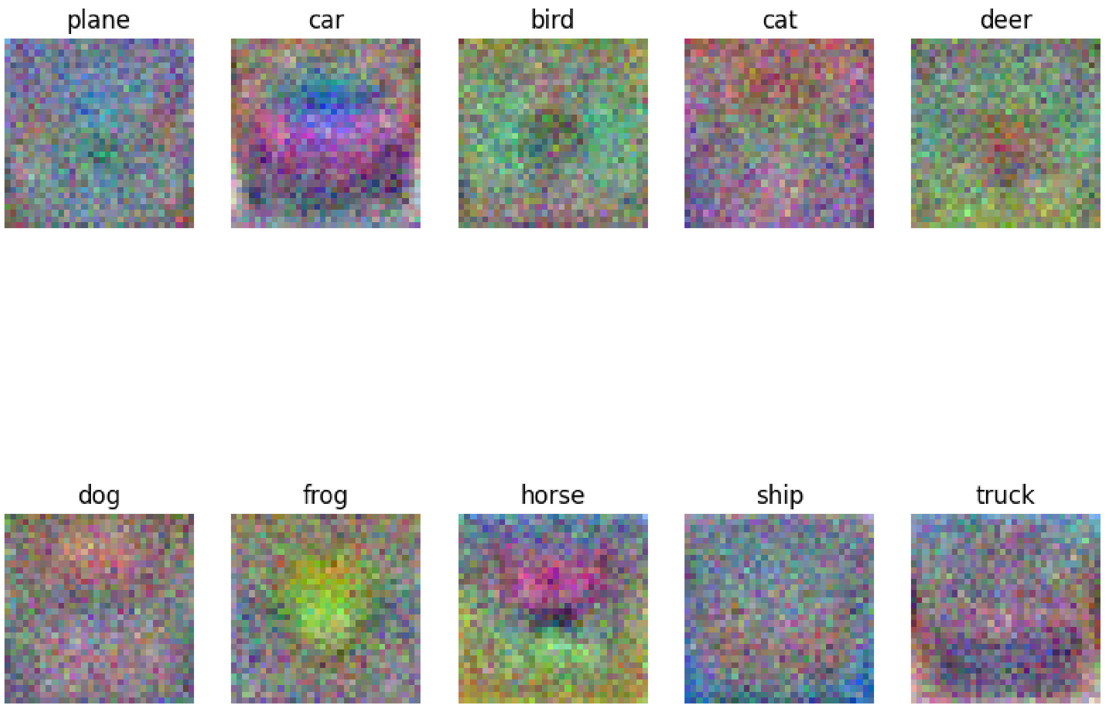
$Your Explanation:$

Suppose we have a new data point for images that can be one of three classes and with a margin of 1 for SVM. Then lets say that the score of the new data point is $[9, 8, 11]$ where class 3 (or 2 for zero-based indexing) is the correct class then when taking the SVM loss we get

$max(0, 9 - 11 + 1) + max(0, 8 - 11 + 1) = 0 + 0 = 0.$

So when we add this datapoint to the data set and the SVM loss doesn't change. While softmax loss will result in some change.

[57]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

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

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

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck