knn

January 23, 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. 'cse493g1/assignments/assignment1/'
FOLDERNAME = 'cse493g1/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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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

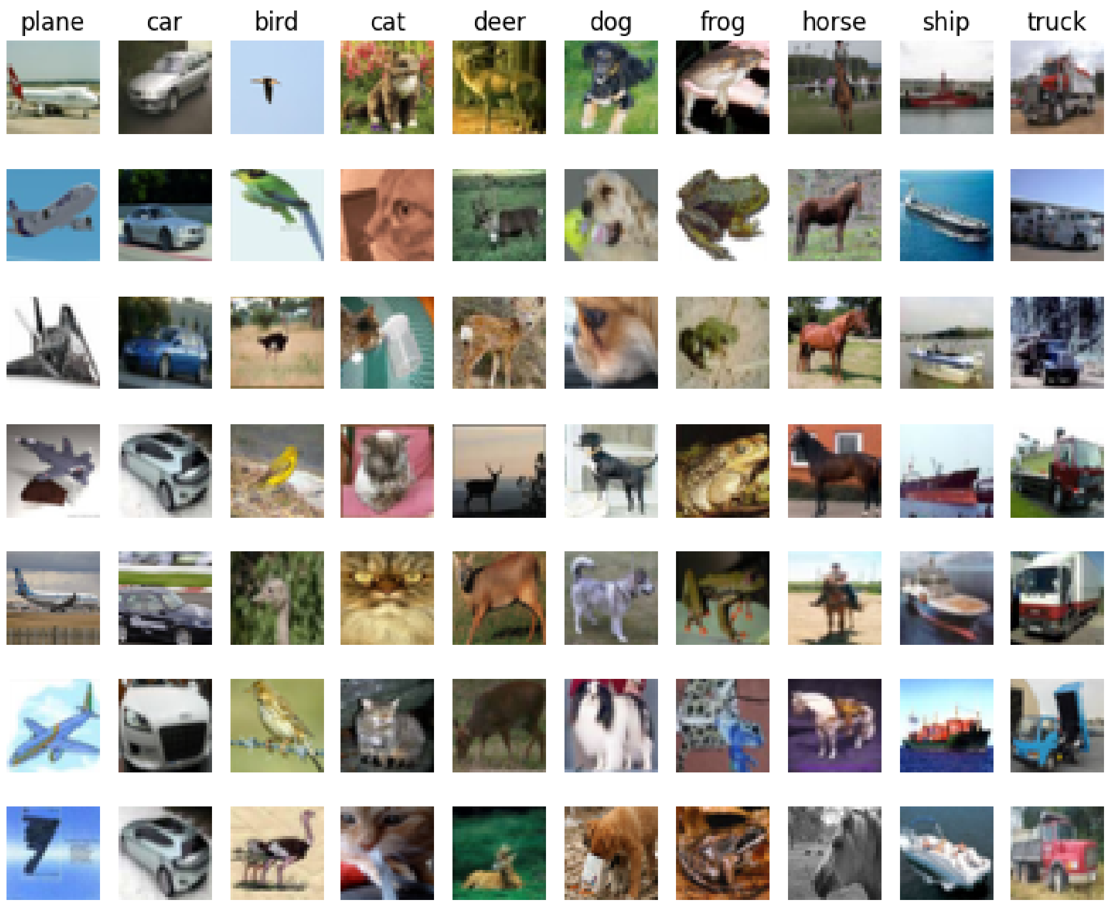
In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

[2]:
```
# 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
```

[3]:
```
# 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,)
```

[4]:
```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()
```



[5]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

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

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

[6]:
```
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.

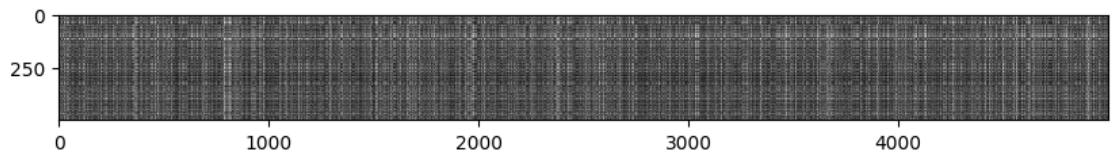
First, open `cse493g1/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

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

[8]:
```
# 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?

$\color{blue}{\textit Your Answer:}$ - Distictly bright rows indicate test images with high distances from most of the training data. - Bright columns indicate training images with high distances from most of the test data.

[9]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

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

[10]:
```
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.

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

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

1. This step subtracts a same amount from every pixel in every image. The relative distances between any pair of points remain unchanged.
2. This step subtracts the per pixel mean across al images. This will not change the relative distance between two points.
3. We know subtracting the mean will not change the performance of the classifier, and dividing by the same number(σ) will only change the scale but will not alter the comparison result.
4. Scaling pixels by different amounts(σ_{ij}) will change the relative distances, and thus affecting the performance of a KNN classifier.
5. This step will change the performance because L1 distance is not rotation-invariant.

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

[12]:
```
# 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
```

[13]:
```
# 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 39.761676 seconds
One loop version took 54.951896 seconds
No loop version took 1.061458 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.

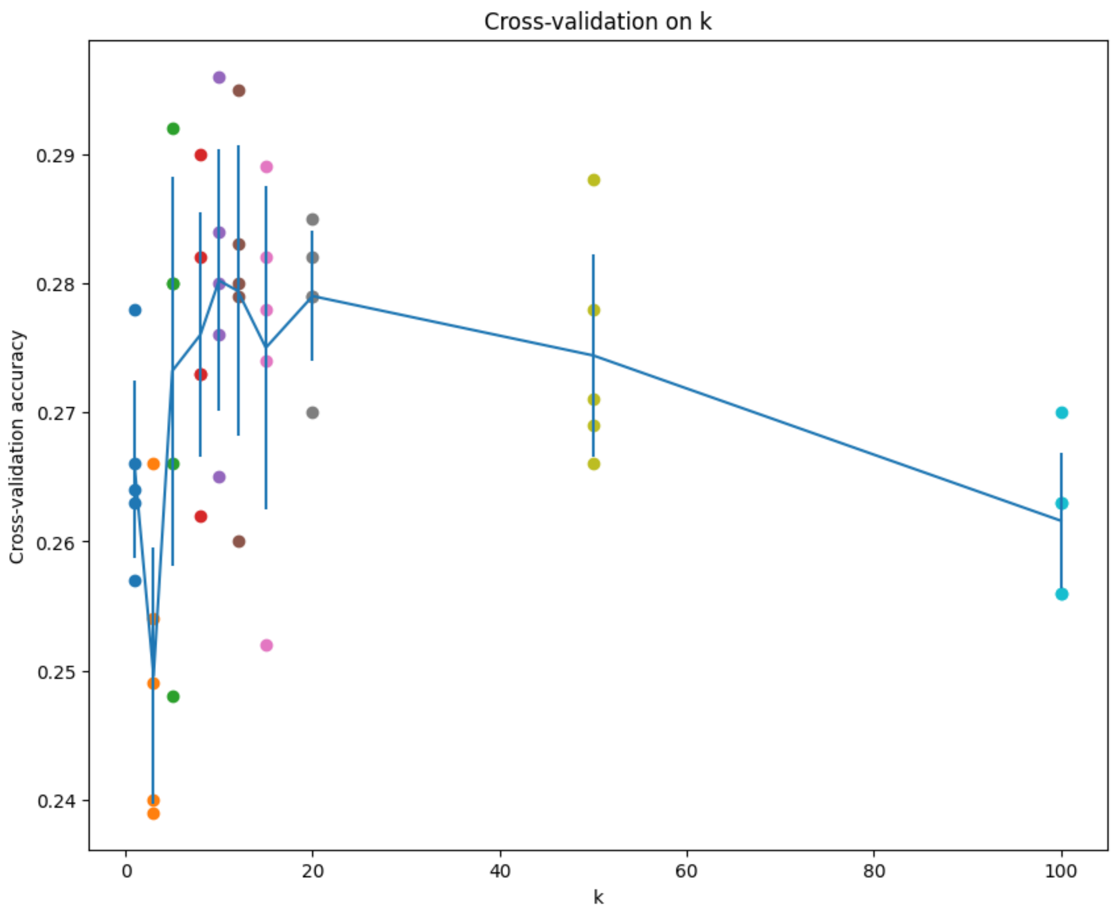
[23]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)

# *****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 i in range(num_folds):
    train_indices = [j for j in range(num_folds) if j != i]

    X_train_i = np.concatenate([X_train_folds[j] for j in train_indices])
    y_train_i = np.concatenate([y_train_folds[j] for j in train_indices])

    classifier.train(X_train_i, y_train_i)

    dists = classifier.compute_distances_no_loops(X_train_folds[i])
    y_test_pred = classifier.predict_labels(dists, k)
    # Compute and print the fraction of correctly predicted examples
    num_correct = np.sum(y_test_pred == y_train_folds[i])
    k_to_accuracies[k].append(float(num_correct) / len(X_train_folds[i]))

# *****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
```

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



[25]:
```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.

Your Answer : 2, 4

Your Explanation : 1. **False** becasue the decision boundary can be quite complex and non-linear, especially with smaller values of k. 2. **True** becasue when k=1, the classifier is perfectly fitted on the training data, meaning there will be no training error. 3. **False** becasue 1-NN can be overfitting on the test data, and might lead to worse performance on test data. 4. **True** becasue k-NN classification requires computing distances from every test image to each point in the training data set. The size of the k-NN classifier will grow with the size of the training set.

svm

January 23, 2024

[4]:
```
# 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/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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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

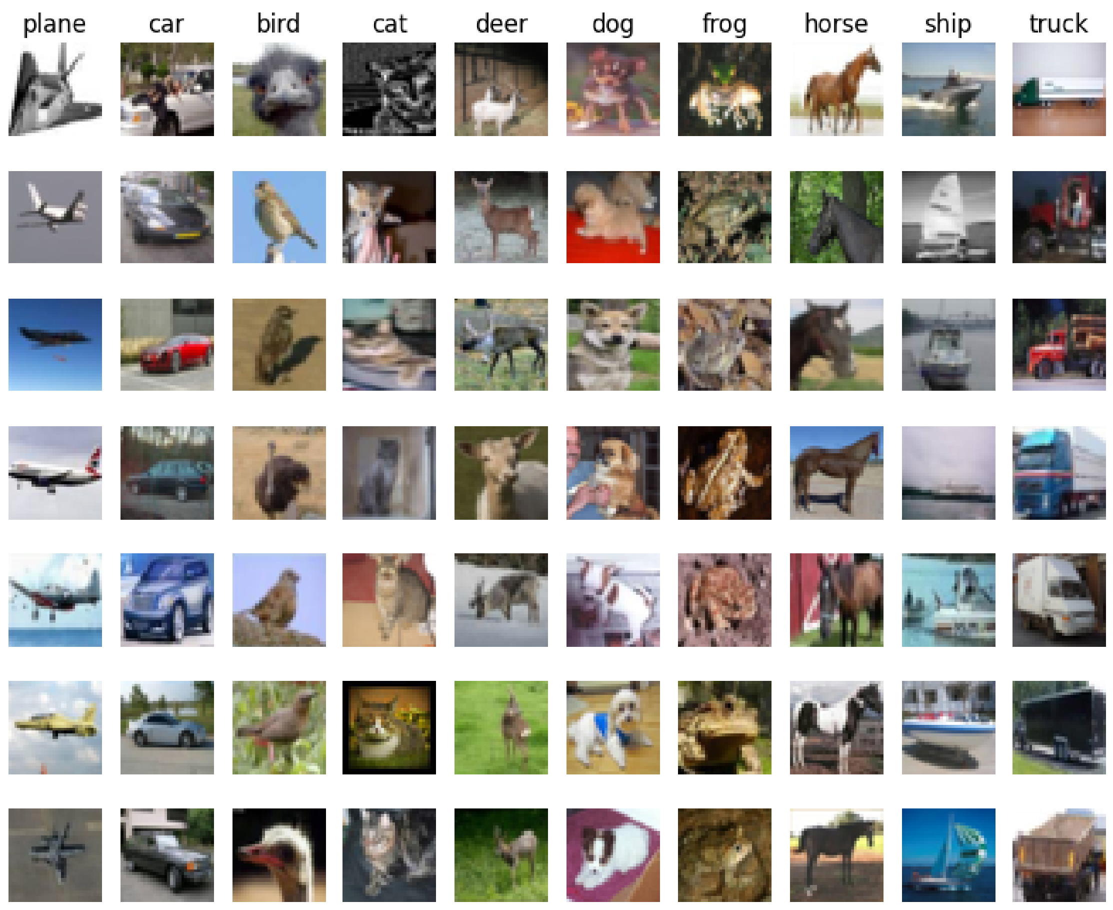
[5]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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

[6]:
```
# 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,)
```

[20]:
```
# 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()
```



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

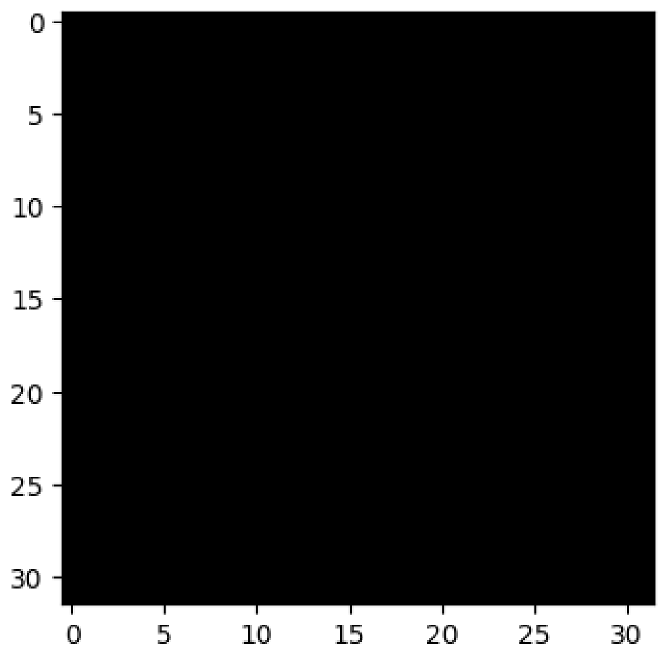
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

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

```
[ 1.18329437e-13 -4.63871300e-14 -3.80850906e-15 -4.01137627e-13
 -1.58515994e-13 -5.21631174e-14  4.76085674e-13  2.89594018e-14
 -1.12851587e-14  2.68177970e-13]
```



```
(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.

[13]:
```python
# 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: 8.592356
```

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:

[14]:
```
# 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: -19.756487 analytic: -19.756487, relative error: 2.495074e-12
numerical: -18.196270 analytic: -18.215399, relative error: 5.253372e-04
numerical: 0.405624 analytic: 0.405624, relative error: 9.885622e-10
numerical: 9.595956 analytic: 9.595956, relative error: 2.061794e-11
numerical: -21.930664 analytic: -21.930664, relative error: 1.175292e-12
numerical: 16.420944 analytic: 16.420944, relative error: 1.265338e-11
numerical: -38.789933 analytic: -38.797312, relative error: 9.510815e-05
numerical: 25.138218 analytic: 25.138218, relative error: 3.970093e-12
numerical: -11.736338 analytic: -11.709735, relative error: 1.134644e-03
numerical: -20.952139 analytic: -20.952139, relative error: 6.589654e-12
numerical: 14.544127 analytic: 14.520325, relative error: 8.189460e-04
numerical: 22.508952 analytic: 22.505359, relative error: 7.982739e-05
numerical: 3.984684 analytic: 3.999492, relative error: 1.854732e-03
numerical: 1.376710 analytic: 1.380444, relative error: 1.354294e-03
numerical: -5.442650 analytic: -5.437788, relative error: 4.468416e-04
numerical: -10.835909 analytic: -10.826608, relative error: 4.293433e-04
numerical: 9.946225 analytic: 9.939248, relative error: 3.508587e-04
numerical: -11.804649 analytic: -11.803363, relative error: 5.444266e-05
numerical: 10.962133 analytic: 10.956554, relative error: 2.545534e-04
numerical: 2.168193 analytic: 2.170930, relative error: 6.309217e-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*

$Your Answer:$

1. *The SVM loss function includes max operations, which are not differentiable at the point where the argument of the max function switches. At such points, the gradient is not defined, leading to discrepancy.*
2. *If the discrepancy is small enough, we do not need to concern too much.*

3. *For example, the gradient of $f(x) = max(0, x)$ when $x = 0$ is not defined.*
4. *A larger margin increases the likelihood that more data points will fall within the margin, potentially increasing the number of non-differentiable points. This can lead to a higher frequency of discrepancies in gradient checks.*

[15]:
```
# 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: 8.592356e+00 computed in 0.070236s
Vectorized loss: 8.592356e+00 computed in 0.003779s
difference: -0.000000
```

[23]:
```
# 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 0.110083s
Vectorized loss and gradient: computed in 0.007174s
difference: 0.000000
```

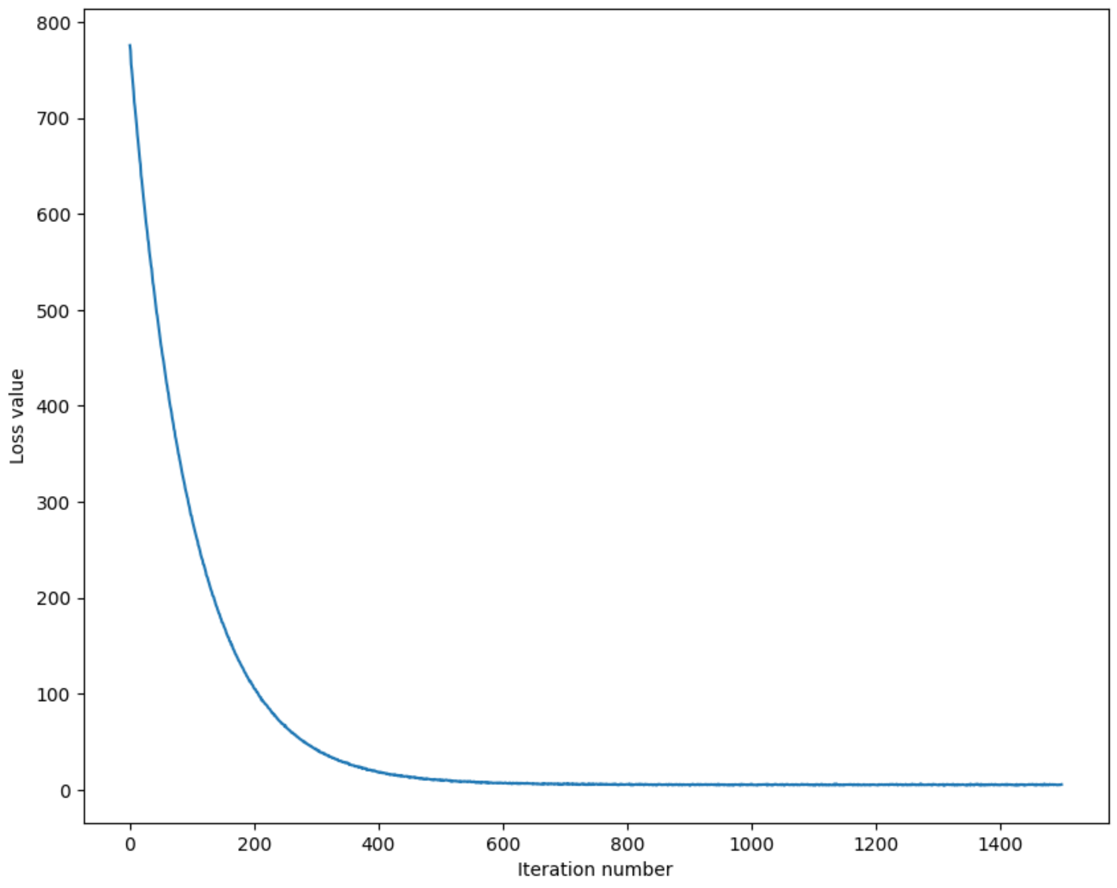
1.2.1 Stochastic Gradient Descent

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`.

[52]:
```
# 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 794.745392
iteration 100 / 1500: loss 289.419877
iteration 200 / 1500: loss 108.006931
iteration 300 / 1500: loss 42.847554
iteration 400 / 1500: loss 19.241875
iteration 500 / 1500: loss 10.806327
iteration 600 / 1500: loss 7.292912
iteration 700 / 1500: loss 6.582946
iteration 800 / 1500: loss 5.554875
iteration 900 / 1500: loss 5.428219
iteration 1000 / 1500: loss 5.136992
iteration 1100 / 1500: loss 5.549500
iteration 1200 / 1500: loss 5.115064
iteration 1300 / 1500: loss 5.351385
iteration 1400 / 1500: loss 5.145123
That took 5.589749s
```

[43]:
```
# 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()
```



[45]:
```
# 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: 0.368020
validation accuracy: 0.365000
```

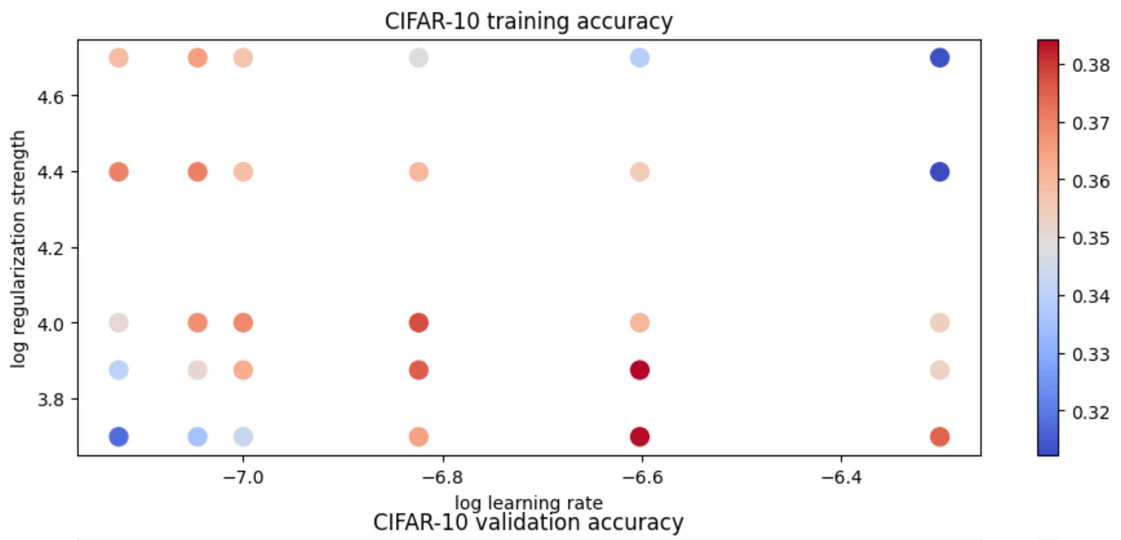
[313]:
```
from re import VERBOSE
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy 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, 5e-5]
# regularization_strengths = [2.5e4, 5e4]
learning_rates = [7.5e-8, 9e-8, 1e-7, 1.5e-7, 2.5e-7, 5e-7]
regularization_strengths = [5e3, 7.5e3, 1e4, 2.5e4, 5e4]

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

for learning_rate in learning_rates:
  for lamda in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate, lamda,
                         num_iters=1000, verbose=False)
    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[(learning_rate, lamda)] = (train_accuracy, val_accuracy)
    if val_accuracy > best_val:
      best_val = val_accuracy
```

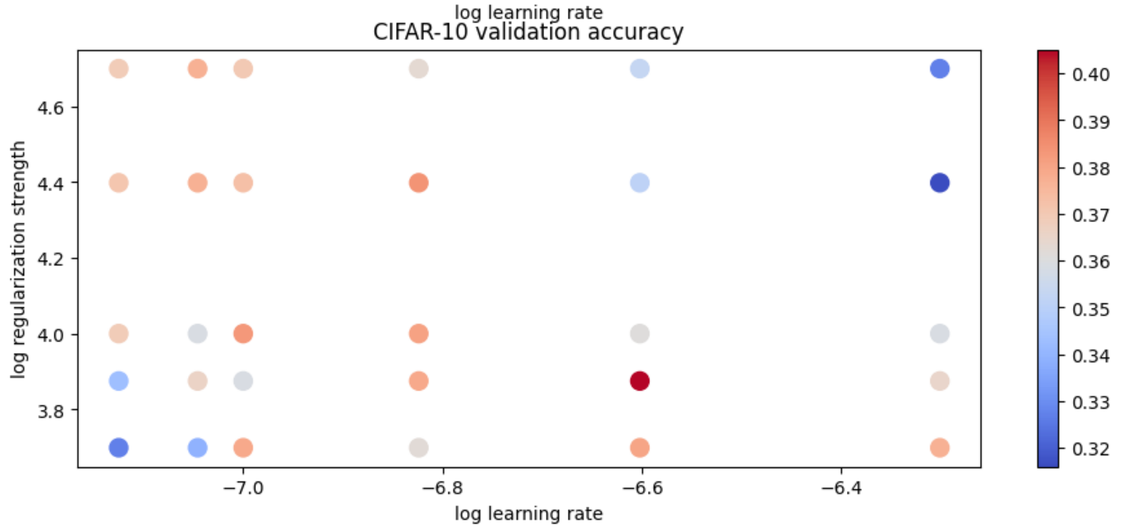
```
        best_svm = svm

# *****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 7.500000e-08 reg 5.000000e+03 train accuracy: 0.317714 val accuracy: 0.327000
lr 7.500000e-08 reg 7.500000e+03 train accuracy: 0.340204 val accuracy: 0.343000
lr 7.500000e-08 reg 1.000000e+04 train accuracy: 0.350898 val accuracy: 0.369000
lr 7.500000e-08 reg 2.500000e+04 train accuracy: 0.370265 val accuracy: 0.371000
lr 7.500000e-08 reg 5.000000e+04 train accuracy: 0.359449 val accuracy: 0.369000
lr 9.000000e-08 reg 5.000000e+03 train accuracy: 0.335490 val accuracy: 0.339000
lr 9.000000e-08 reg 7.500000e+03 train accuracy: 0.351571 val accuracy: 0.366000
lr 9.000000e-08 reg 1.000000e+04 train accuracy: 0.368143 val accuracy: 0.359000
lr 9.000000e-08 reg 2.500000e+04 train accuracy: 0.370449 val accuracy: 0.377000
lr 9.000000e-08 reg 5.000000e+04 train accuracy: 0.365571 val accuracy: 0.377000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.342878 val accuracy: 0.379000
lr 1.000000e-07 reg 7.500000e+03 train accuracy: 0.362592 val accuracy: 0.359000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.369020 val accuracy: 0.383000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.358265 val accuracy: 0.373000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.357388 val accuracy: 0.370000
lr 1.500000e-07 reg 5.000000e+03 train accuracy: 0.364816 val accuracy: 0.362000
lr 1.500000e-07 reg 7.500000e+03 train accuracy: 0.375510 val accuracy: 0.379000
lr 1.500000e-07 reg 1.000000e+04 train accuracy: 0.377776 val accuracy: 0.381000
lr 1.500000e-07 reg 2.500000e+04 train accuracy: 0.360000 val accuracy: 0.384000
lr 1.500000e-07 reg 5.000000e+04 train accuracy: 0.347816 val accuracy: 0.363000
lr 2.500000e-07 reg 5.000000e+03 train accuracy: 0.383531 val accuracy: 0.380000
lr 2.500000e-07 reg 7.500000e+03 train accuracy: 0.384306 val accuracy: 0.405000
lr 2.500000e-07 reg 1.000000e+04 train accuracy: 0.360367 val accuracy: 0.361000
lr 2.500000e-07 reg 2.500000e+04 train accuracy: 0.355041 val accuracy: 0.351000
lr 2.500000e-07 reg 5.000000e+04 train accuracy: 0.338939 val accuracy: 0.353000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.375000 val accuracy: 0.377000
lr 5.000000e-07 reg 7.500000e+03 train accuracy: 0.352878 val accuracy: 0.365000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.353653 val accuracy: 0.359000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.312184 val accuracy: 0.316000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.312980 val accuracy: 0.327000
best validation accuracy achieved during cross-validation: 0.405000
```

[314]:
```
# Visualize the cross-validation results
import math
```

```python
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()
```



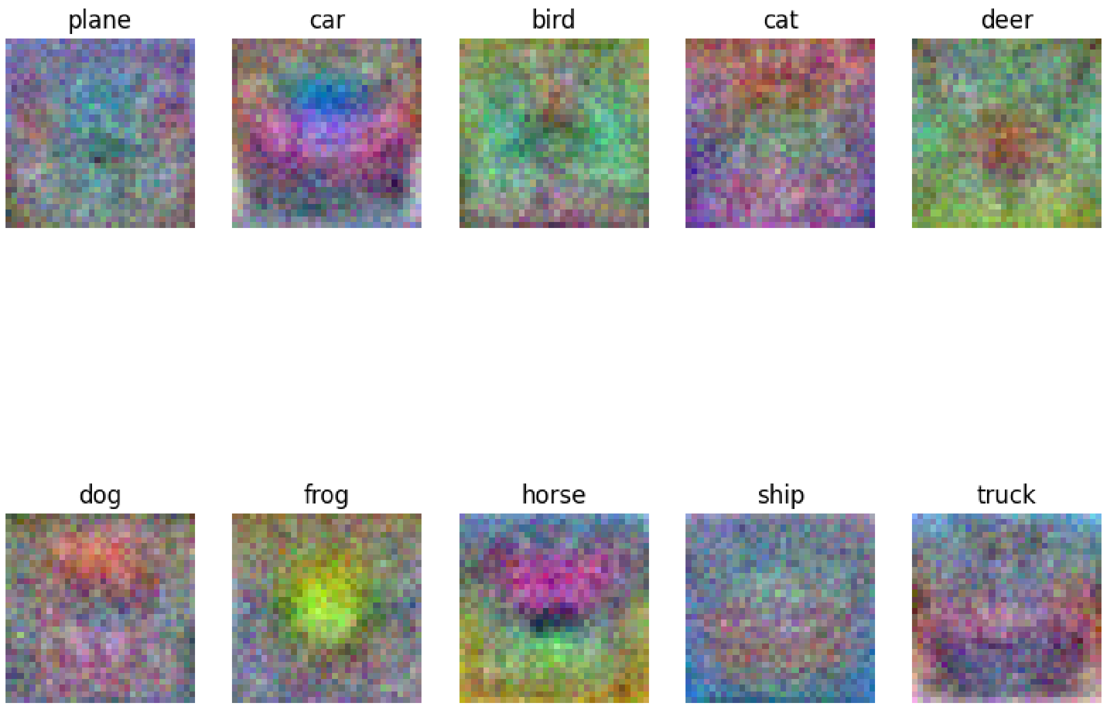


[315]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.387000
```

[316]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
 ↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
```

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

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



Inline question 2

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

YourAnswer : They look like an average representation for each class. For example, the visualized SVM weights for the car class look like a blurry red/blue car. They look like this because the SVM model learns from the data set, and the weights capture and store the most distinguishing features for each class.

softmax

January 23, 2024

[13]:
```
# 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/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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$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/assignments/assignment1/cse493g1/datasets
/content/drive/My Drive/cse493g1/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

[14]:
```
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
```

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

[21]:
```
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`.

[40]:
```python
# 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: 2.344864
sanity check: 2.302585
```

Inline Question 1

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

$\color{blue}{\textit{Your Answer :}}$ *The weights W are initialized randomly with very small numbers since we scale it with* 0.0001. *Therefore,* e^{s_j} *for any j will be close to 1. Since there are 10 classes, the probability assigned to the correct class for each example is about 1/10 or 0.1. Thus, the loss for a single example will be* $-np.log(0.1)$.

[58]:
```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 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: -0.287292 analytic: -0.287292, relative error: 1.368844e-07
numerical: -2.133012 analytic: -2.133012, relative error: 9.928172e-09
numerical: 3.486579 analytic: 3.486579, relative error: 1.902934e-08
numerical: -0.638391 analytic: -0.638391, relative error: 2.786998e-09
numerical: 0.466032 analytic: 0.466032, relative error: 4.656824e-08
numerical: -4.446329 analytic: -4.446329, relative error: 1.092462e-08
numerical: 1.237219 analytic: 1.237219, relative error: 2.635033e-08
numerical: 2.467124 analytic: 2.467124, relative error: 3.263277e-08
numerical: -0.562054 analytic: -0.562054, relative error: 7.212781e-08
numerical: 0.655912 analytic: 0.655912, relative error: 3.587182e-08
numerical: -2.553956 analytic: -2.553956, relative error: 3.111392e-08
numerical: 3.630114 analytic: 3.630114, relative error: 2.671125e-08
numerical: -0.252364 analytic: -0.252364, relative error: 1.783627e-07
numerical: 0.230186 analytic: 0.230186, relative error: 1.016558e-07
numerical: 2.768403 analytic: 2.768403, relative error: 1.784648e-08
numerical: -1.180722 analytic: -1.180722, relative error: 1.136056e-08
numerical: -0.300884 analytic: -0.300884, relative error: 1.157756e-07
numerical: 1.369998 analytic: 1.369998, relative error: 2.020482e-09
numerical: 1.545710 analytic: 1.545710, relative error: 2.800696e-08
numerical: -2.271695 analytic: -2.271695, relative error: 5.046062e-09
```

[69]:
```
# 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: 2.344864e+00 computed in 0.179983s
vectorized loss: 2.344864e+00 computed in 0.015619s
```

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

[71]:
```
# 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 = [7.5e-8, 1e-7, 5e-7]
regularization_strengths = [1e4, 2.5e4, 5e4]

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

for lr in learning_rates:
  for reg in regularization_strengths:
    softMax = Softmax()
    softMax.train(X_train, y_train, lr, reg,
                         num_iters=1000, verbose=True)
    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[(lr, reg)] = (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)
```

```
iteration 0 / 1000: loss 313.276017
iteration 100 / 1000: loss 231.145464
iteration 200 / 1000: loss 171.232834
iteration 300 / 1000: loss 127.262437
iteration 400 / 1000: loss 94.608101
iteration 500 / 1000: loss 70.409052
iteration 600 / 1000: loss 52.656932
iteration 700 / 1000: loss 39.589237
iteration 800 / 1000: loss 29.704645
iteration 900 / 1000: loss 22.421089
iteration 0 / 1000: loss 783.215619
iteration 100 / 1000: loss 369.312634
iteration 200 / 1000: loss 174.917883
iteration 300 / 1000: loss 83.505197
iteration 400 / 1000: loss 40.360040
iteration 500 / 1000: loss 20.165467
iteration 600 / 1000: loss 10.677576
iteration 700 / 1000: loss 6.064199
iteration 800 / 1000: loss 3.955036
iteration 900 / 1000: loss 2.996863
iteration 0 / 1000: loss 1533.554724
iteration 100 / 1000: loss 340.862713
iteration 200 / 1000: loss 77.141256
iteration 300 / 1000: loss 18.750570
iteration 400 / 1000: loss 5.789590
iteration 500 / 1000: loss 3.006853
iteration 600 / 1000: loss 2.337328
iteration 700 / 1000: loss 2.194684
iteration 800 / 1000: loss 2.153675
iteration 900 / 1000: loss 2.192035
iteration 0 / 1000: loss 315.294281
iteration 100 / 1000: loss 210.514417
iteration 200 / 1000: loss 141.313942
iteration 300 / 1000: loss 95.171310
iteration 400 / 1000: loss 64.146748
iteration 500 / 1000: loss 43.489463
iteration 600 / 1000: loss 29.803036
iteration 700 / 1000: loss 20.621932
iteration 800 / 1000: loss 14.391803
iteration 900 / 1000: loss 10.253224
iteration 0 / 1000: loss 766.953416
iteration 100 / 1000: loss 281.283787
iteration 200 / 1000: loss 104.205706
iteration 300 / 1000: loss 39.433886
iteration 400 / 1000: loss 15.734797
iteration 500 / 1000: loss 7.122617
iteration 600 / 1000: loss 3.960338
iteration 700 / 1000: loss 2.750592
iteration 800 / 1000: loss 2.274283
iteration 900 / 1000: loss 2.192539
iteration 0 / 1000: loss 1536.944243
iteration 100 / 1000: loss 206.926501
iteration 200 / 1000: loss 29.545975
iteration 300 / 1000: loss 5.799140
iteration 400 / 1000: loss 2.647989
iteration 500 / 1000: loss 2.182045
iteration 600 / 1000: loss 2.133431
iteration 700 / 1000: loss 2.211562
iteration 800 / 1000: loss 2.155845
iteration 900 / 1000: loss 2.147599
iteration 0 / 1000: loss 306.811760
iteration 100 / 1000: loss 42.082063
iteration 200 / 1000: loss 7.391586
iteration 300 / 1000: loss 2.735437
iteration 400 / 1000: loss 2.111601
iteration 500 / 1000: loss 2.013753
iteration 600 / 1000: loss 2.004462
iteration 700 / 1000: loss 2.020943
iteration 800 / 1000: loss 2.010355
iteration 900 / 1000: loss 1.995354
iteration 0 / 1000: loss 776.062224
iteration 100 / 1000: loss 6.926944
iteration 200 / 1000: loss 2.093072
iteration 300 / 1000: loss 2.111707
iteration 400 / 1000: loss 2.149608
iteration 500 / 1000: loss 2.078270
iteration 600 / 1000: loss 2.052848
iteration 700 / 1000: loss 2.095065
iteration 800 / 1000: loss 2.105798
iteration 900 / 1000: loss 2.086939
iteration 0 / 1000: loss 1562.457505
iteration 100 / 1000: loss 2.237177
iteration 200 / 1000: loss 2.140635
iteration 300 / 1000: loss 2.112624
iteration 400 / 1000: loss 2.151101
iteration 500 / 1000: loss 2.130567
iteration 600 / 1000: loss 2.152658
iteration 700 / 1000: loss 2.165691
iteration 800 / 1000: loss 2.113372
iteration 900 / 1000: loss 2.190204
```

```
lr 7.500000e-08 reg 1.000000e+04 train accuracy: 0.304490 val accuracy: 0.313000
lr 7.500000e-08 reg 2.500000e+04 train accuracy: 0.326510 val accuracy: 0.341000
lr 7.500000e-08 reg 5.000000e+04 train accuracy: 0.302163 val accuracy: 0.318000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.329224 val accuracy: 0.322000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.328714 val accuracy: 0.333000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.300469 val accuracy: 0.320000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.342531 val accuracy: 0.358000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.326592 val accuracy: 0.339000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.300469 val accuracy: 0.322000
best validation accuracy achieved during cross-validation: 0.358000
```

[72]:
```
# 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: 0.355000
```

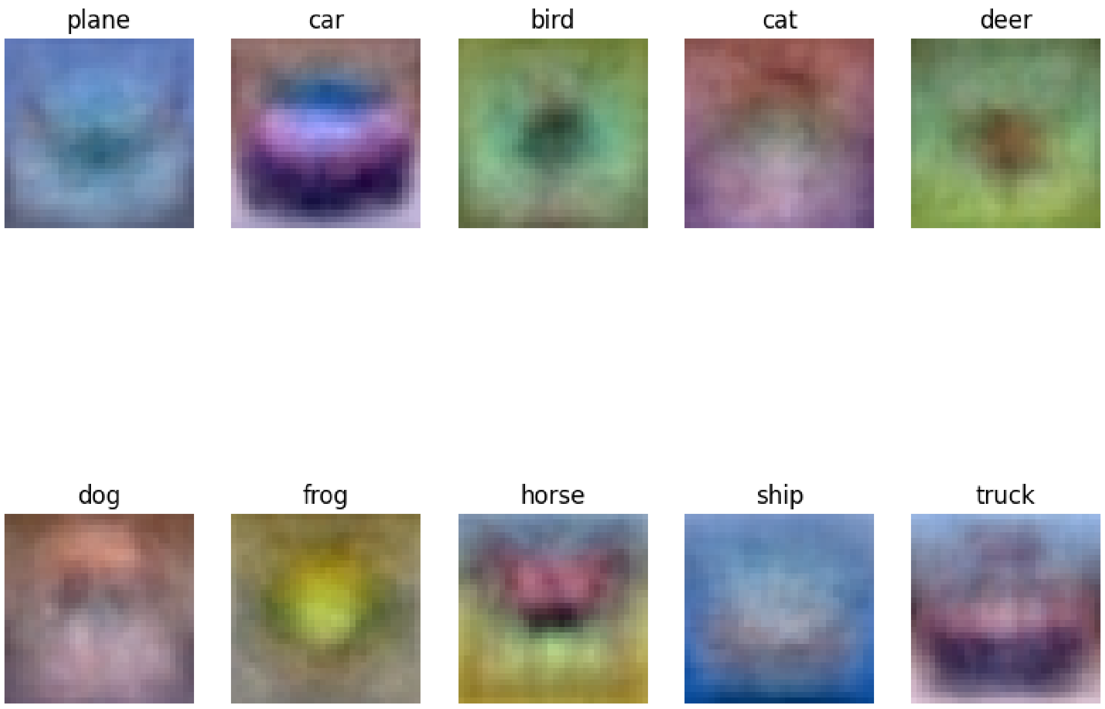
Inline Question 2 - *True or False*

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 : *True*

Your Explanation : *For SVM, it is possible to have a zero loss on a correctly classified data point, so the overall loss will not be changed.*
For Softmax, the loss is calculated based on assigned probability. The only time that the loss of the new data point can be zero is when the correct class has a probability of 1, which is not likely to happen using real world data.

[73]:
```
# 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