knn

September 24, 2019

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.

[1]:
```
# Run some setup code for this notebook.

from __future__ import print_function
import random
import numpy as np
from cs682.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
```

[2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/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,)
```

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

[4]:
```
# 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]
```

[5]:
```
# 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 cs682.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

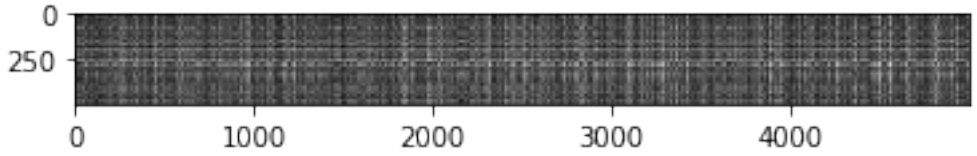
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

First, open `cs682/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 cs682/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 visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

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

Your Answer: 1. Distinctly bright rows mean that some of the test data set are very different from the entire training data set. 2. Distinctly bright columns mean that some of the training data set are very different from the entire test set.

[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 other distance metrics such as L1 distance. The performance of a Nearest Neighbor classifier that uses L1 distance will not change if (Select all that apply.): 1. The data is preprocessed by subtracting the mean. 2. The data is preprocessed by subtracting the mean and dividing by the standard deviation. 3. The coordinate axes for the data are rotated. 4. None of the above.

Your Answer: 1, 2, 3

Your explanation: since the above operations would equally modify the coordinates of the points being compared, keeping the L1 distance same, so the performance of the Nearest Neighbour classifier would not change.

[11]:
```
# 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('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')
```

```
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('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')
```

```
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
```

```
Two loop version took 28.459913 seconds
One loop version took 30.967178 seconds
No loop version took 0.143717 seconds
```

1.0.1 Cross-validation

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

[14]:
```
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.                                #
################################################################################
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# 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.                               #
################################################################################
for k in k_choices:
```

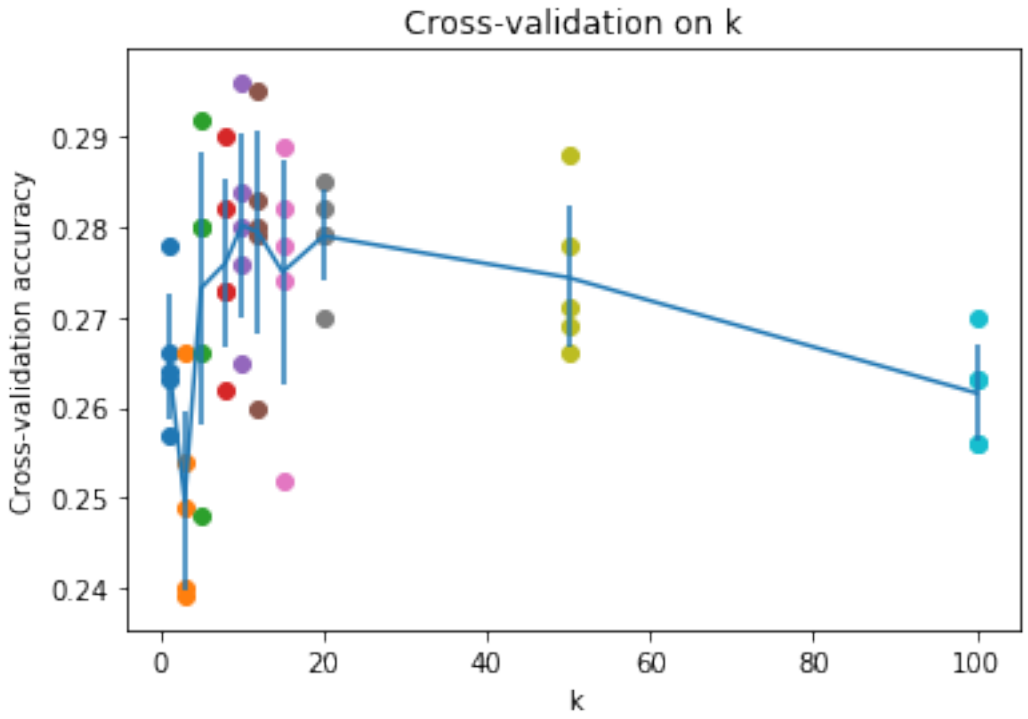
```
    k_to_accuracies[k]=[]
    for i in range(num_folds):
        #preparing the training data set
        xtrain = np.vstack(X_train_folds[:i]+X_train_folds[i+1:])
        ytrain = np.concatenate(y_train_folds[:i]+y_train_folds[i+1:],0)

        #training the classifier
        classifier = KNearestNeighbor()
        classifier.train(xtrain, ytrain)

        #validation set
        validation_set_x = X_train_folds[i]
        validation_set_y = y_train_folds[i]

        #predicting
        num_test_val = validation_set_x.shape[0]
        y_test_pred = classifier.predict(validation_set_x, k)
        num_correct = np.sum(y_test_pred == validation_set_y)
        accuracy = float(num_correct) / num_test_val
        k_to_accuracies[k].append(accuracy)

################################################################################
#                                END OF YOUR CODE                              #
################################################################################

# Print out the computed accuracies
best_k = -1
best_k_acc = -1
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
```

[15]:
```
# 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()
print(accuracies_mean)
```




```
[0.2656 0.2496 0.2732 0.276  0.2802 0.2794 0.275  0.279  0.2744 0.2616]
```

[16]:
```
# 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.
idx=np.argmax(accuracies_mean)
best_k=k_choices[idx]

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 training error of a 1-NN will always be better than that of 5-NN. 2. The test error of a 1-NN will always be better than that of a 5-NN. 3. The decision boundary of the k-NN classifier is linear. 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: 1

Your explanation: 1. Since 1-NN has a 100% accuracy as for each node the classifier would choose itself everytime. As opposed, the k-NN for any k>1 would have accuracy <=100 as the more than one node apart from the node itself is involved in the decision making process 2. This does not hold true. The test error of 5-NN is likely to be performing better as Peter Stone proved for large datasets and as k->infinity the result of the classifier becomes near perfect. 3. This does not hold true as we cannot represent the k-NN using a linear mathematical function, and can be easily visualised by taking a small example. 4. This does not apply as the time needed to classify a test for any k-NN classifier is independent of the training set size, since it is a no-op, and the actual time is computation happens when the classification occurs.

svm

September 24, 2019

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

[1]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs682.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

[2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/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,)
```

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

[4]:
```
# 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)
```

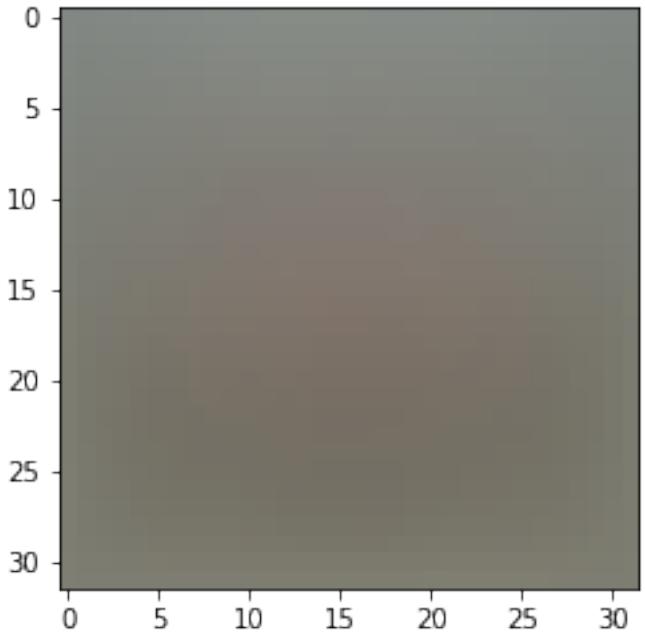
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

[6]:
```
# 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()
```

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



[7]:
```
# 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
```

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

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

1.2 SVM Classifier

Your code for this section will all be written inside **cs682/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.

[9]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs682.classifiers.linear_svm import svm_loss_naive
```

```
import time
import numpy as np

# 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: 9.316212
```

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

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

[10]:
```
# 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 cs682.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: -2.495961 analytic: -2.495961, relative error: 2.166647e-08
numerical: -7.145851 analytic: -7.145851, relative error: 4.197481e-09
numerical: -9.625754 analytic: -9.625754, relative error: 3.593288e-09
numerical: -11.534530 analytic: -11.534530, relative error: 4.975263e-10
numerical: -14.480223 analytic: -14.480223, relative error: 8.038915e-11
numerical: -9.554057 analytic: -9.554057, relative error: 4.782413e-09
numerical: 0.323823 analytic: 0.323824, relative error: 8.224549e-08
numerical: 21.929613 analytic: 21.929613, relative error: 9.039412e-10
numerical: 4.173760 analytic: 4.173760, relative error: 1.209036e-09
numerical: 39.665447 analytic: 39.665447, relative error: 1.876615e-10
```

```
numerical: -17.813928 analytic: -17.813928, relative error: 1.127070e-09
numerical: 27.082425 analytic: 27.082425, relative error: 1.529059e-09
numerical: 0.185539 analytic: 0.185539, relative error: 4.520337e-08
numerical: 1.324953 analytic: 1.324953, relative error: 2.210656e-08
numerical: -13.111219 analytic: -13.111219, relative error: 1.491303e-09
numerical: -23.681348 analytic: -23.681348, relative error: 2.390839e-10
numerical: -20.007515 analytic: -20.007515, relative error: 1.056836e-09
numerical: 14.290475 analytic: 14.290475, relative error: 2.105156e-09
numerical: 9.946344 analytic: 9.946344, relative error: 2.128535e-09
numerical: -10.695458 analytic: -10.695457, relative error: 2.850588e-09
```

1.2.1 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: The discrepancy is caused by the granularity of our mumeric differential method. If we increase the granularity from the current 10^-5 to say 10^-6 we would observe that the relative error reduces further. If we take only single dimension, then by plotting the graph of the function max(0,s_other - s_correct), the point where s_other - s_correct becomes 0 is non differentiable, as there is a sharp turn there.

[11]:
```
# 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 cs682.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: %.18f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.316212e+00 computed in 0.095964s
Vectorized loss: 9.316212e+00 computed in 0.092769s
difference: 0.000000000000012434
```

[12]:
```
# 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: %.18f' % difference)
```

```
Naive loss and gradient: computed in 0.093284s
Vectorized loss and gradient: computed in 0.090564s
difference: 0.000000000005662041
```

1.2.2 Stochastic Gradient Descent

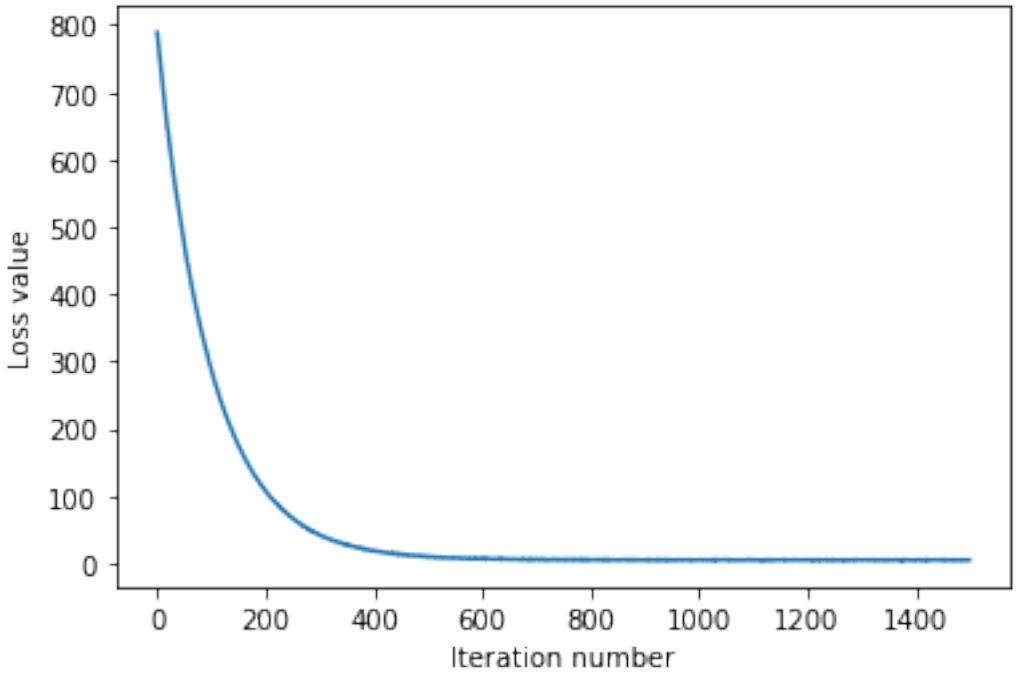
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

[13]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs682.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 789.093102
iteration 100 / 1500: loss 286.483210
iteration 200 / 1500: loss 107.735332
iteration 300 / 1500: loss 42.349344
iteration 400 / 1500: loss 18.897371
iteration 500 / 1500: loss 10.134229
iteration 600 / 1500: loss 6.692520
iteration 700 / 1500: loss 6.462065
iteration 800 / 1500: loss 5.377599
iteration 900 / 1500: loss 5.656864
iteration 1000 / 1500: loss 5.631967
```

```
iteration 1100 / 1500: loss 5.317397
iteration 1200 / 1500: loss 5.584095
iteration 1300 / 1500: loss 5.746603
iteration 1400 / 1500: loss 4.907900
That took 55.218041s
```

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



[15]:
```
# 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: 0.370837
validation accuracy: 0.390000
```

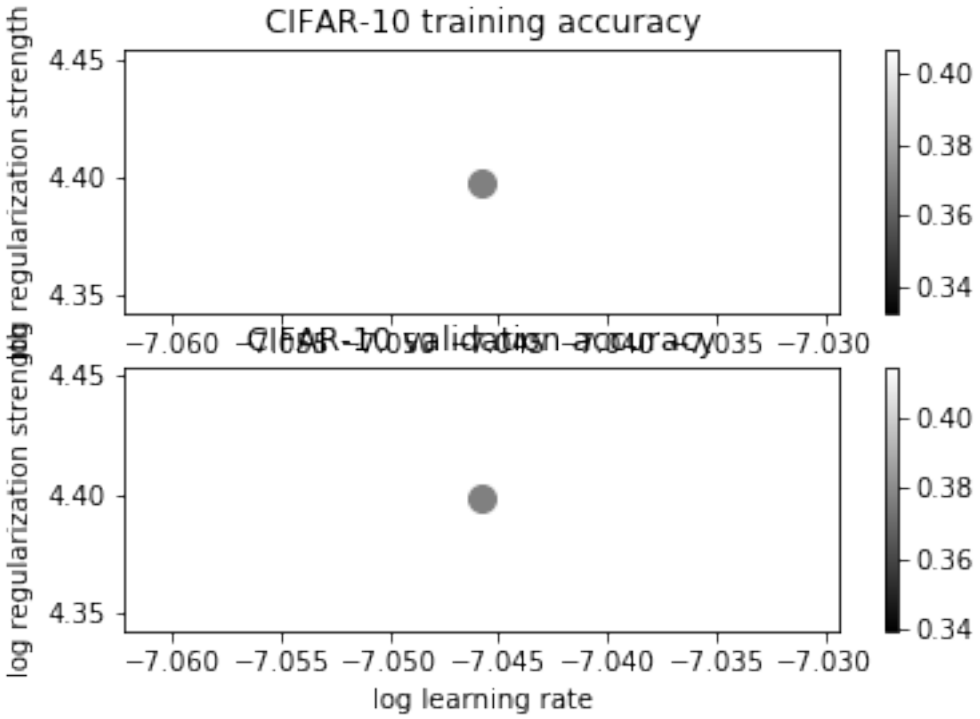
[16]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.


# 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.                                      ␣
 ↪#
################################################################################

lrs = np.arange(1e-7,5e-5,1e-4).tolist()
regs = np.arange(2.5e4,5e4,500).tolist()

lrs = [1e-4,1e-5,1e-6,1e-7,1e-8]
```

```
regs = [1000,5000,10000,25000,50000,75000]

lrs = [0.9e-7]
regs = [2.5e4]

for rg in regs:
    for lr in lrs:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rg,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)
        if best_val<val_acc:
            best_val=val_acc
            best_svm=svm
        results[(lr,rg)] = (train_acc,val_acc)
        print("reg: %d, lr %f, val_acc %f"%(rg,lr,val_acc))


################################################################################
#                               END OF YOUR CODE                               #
################################################################################

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

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

```
iteration 0 / 1500: loss 789.778067
iteration 100 / 1500: loss 317.880999
iteration 200 / 1500: loss 131.572468
iteration 300 / 1500: loss 56.491198
iteration 400 / 1500: loss 25.750803
iteration 500 / 1500: loss 13.531277
iteration 600 / 1500: loss 8.461321
iteration 700 / 1500: loss 6.501563
iteration 800 / 1500: loss 6.531085
iteration 900 / 1500: loss 5.352725
iteration 1000 / 1500: loss 5.411502
iteration 1100 / 1500: loss 5.196632
```

```
iteration 1200 / 1500: loss 5.157001
iteration 1300 / 1500: loss 5.492002
iteration 1400 / 1500: loss 5.343645
reg: 25000, lr 0.000000, val_acc 0.377000
lr 9.000000e-08 reg 2.500000e+04 train accuracy: 0.369429 val accuracy: 0.377000
best validation accuracy achieved during cross-validation: 0.377000
```

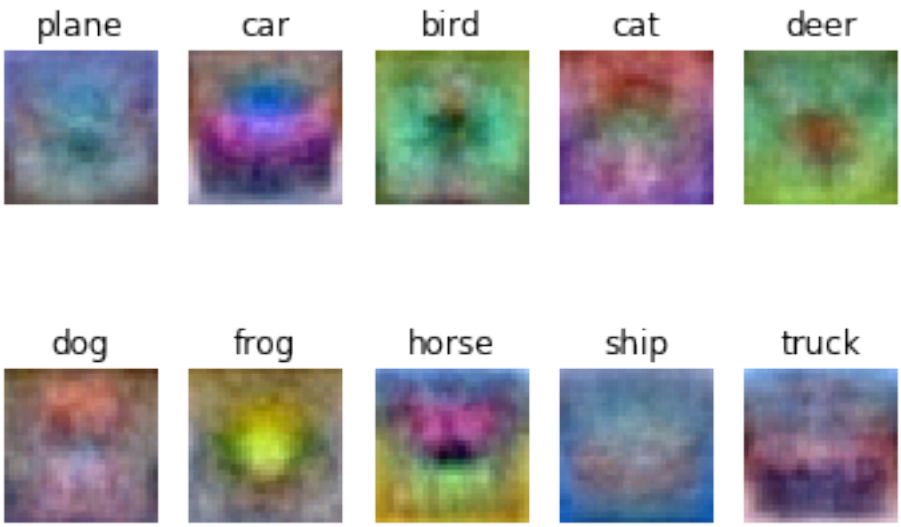
[17]:
```python
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
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)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



[18]:
```
# 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: 0.355000
```

[19]:
```
# 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])
```



1.2.3 Inline question 2:

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

Your answer: The visualizations of the weights look like "rough blurry negative images" of their respective classes. Now ideally the loss function for a perfect svm, would be an ideal near-negative of the image such that the product sum (X.W) with the weights is 0. So, given the accuracy, the appearance should be close to a rough blurry negative image.

softmax

September 24, 2019

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

[1]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.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
```

[2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
```

```
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

    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


# 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

# 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 **cs682/classifiers/softmax.py**.

[3]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs682/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs682.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: 2.390988
sanity check: 2.302585
```

1.2 Inline Question 1:

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

Your answer: This is because we know that e^0=1 since our weights are very close to zero, Sj~0, where Sj is the score for the jth class for say an input Xi. Now consider an input the softmax loss function would be of the form = -log($e^{0/(e}$0+e^{0+e}0+e^{0+e}0... 10 times)) since we have 10 classes. which reduces to -log(1/10) = -log(0.1) Now this will be across all inputs as well, so the mean or the total loss would be 1/N * (N * (-log(0.1))) = -(log(0.1))

[4]:
```
# 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 cs682.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, 100)

# 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, 100)
```

```
numerical: -2.531355 analytic: -2.531355, relative error: 6.946173e-10
numerical: 0.810639 analytic: 0.810639, relative error: 4.468139e-09
numerical: 1.924442 analytic: 1.924442, relative error: 7.558841e-10
numerical: -1.203214 analytic: -1.203214, relative error: 3.405034e-09
numerical: 3.341637 analytic: 3.341637, relative error: 6.289752e-10
numerical: -0.619332 analytic: -0.619332, relative error: 8.329706e-10
numerical: 4.110635 analytic: 4.110635, relative error: 5.582156e-10
numerical: -0.529608 analytic: -0.529608, relative error: 9.599540e-09
numerical: -2.334123 analytic: -2.334123, relative error: 3.660917e-11
numerical: -2.922881 analytic: -2.922881, relative error: 4.009261e-09
numerical: -0.258704 analytic: -0.258704, relative error: 8.082600e-09
numerical: 1.631413 analytic: 1.631413, relative error: 2.244813e-09
numerical: -0.141468 analytic: -0.141468, relative error: 1.776175e-08
numerical: -4.533192 analytic: -4.533192, relative error: 7.301352e-10
numerical: 1.604748 analytic: 1.604748, relative error: 4.175817e-09
numerical: 1.304837 analytic: 1.304838, relative error: 3.130646e-09
numerical: -1.072982 analytic: -1.072982, relative error: 3.368949e-09
numerical: -0.112630 analytic: -0.112630, relative error: 3.885262e-08
numerical: -3.363442 analytic: -3.363442, relative error: 1.561145e-11
numerical: 2.591144 analytic: 2.591144, relative error: 1.313026e-10
numerical: -3.271348 analytic: -3.271348, relative error: 5.643594e-10
numerical: 0.910816 analytic: 0.910816, relative error: 3.232046e-10
numerical: 0.178701 analytic: 0.178701, relative error: 6.113484e-08
numerical: -2.898350 analytic: -2.898350, relative error: 1.242306e-09
numerical: -4.511605 analytic: -4.511605, relative error: 2.193802e-10
```

```
numerical: -4.692728 analytic: -4.692728, relative error: 1.074728e-09
numerical: -1.692584 analytic: -1.692584, relative error: 7.460051e-10
numerical: 1.558374 analytic: 1.558374, relative error: 4.770303e-09
numerical: 2.931170 analytic: 2.931170, relative error: 1.184213e-09
numerical: -0.078489 analytic: -0.078489, relative error: 5.380381e-09
numerical: -2.022956 analytic: -2.022956, relative error: 9.445259e-10
numerical: -0.554712 analytic: -0.554712, relative error: 8.302725e-09
numerical: -0.150180 analytic: -0.150180, relative error: 1.926582e-09
numerical: 0.186099 analytic: 0.186099, relative error: 1.344809e-08
numerical: -1.111407 analytic: -1.111407, relative error: 3.422999e-11
numerical: 1.225024 analytic: 1.225024, relative error: 3.360845e-09
numerical: 2.079053 analytic: 2.079053, relative error: 2.594689e-09
numerical: 0.753875 analytic: 0.753875, relative error: 8.794907e-09
numerical: 0.532034 analytic: 0.532034, relative error: 5.040419e-09
numerical: 1.719698 analytic: 1.719698, relative error: 5.553571e-10
numerical: -0.729944 analytic: -0.729944, relative error: 2.229495e-09
numerical: -3.633529 analytic: -3.633529, relative error: 1.118926e-09
numerical: 0.006526 analytic: 0.006526, relative error: 6.848492e-07
numerical: -0.079615 analytic: -0.079615, relative error: 2.019549e-08
numerical: -0.391065 analytic: -0.391064, relative error: 3.207531e-08
numerical: -1.392199 analytic: -1.392199, relative error: 4.667744e-09
numerical: 0.893272 analytic: 0.893272, relative error: 4.226353e-09
numerical: 2.682547 analytic: 2.682547, relative error: 1.324450e-09
numerical: 2.320033 analytic: 2.320033, relative error: 3.782734e-10
numerical: -4.903229 analytic: -4.903229, relative error: 1.313114e-09
numerical: 2.757075 analytic: 2.757075, relative error: 1.321688e-09
numerical: 2.452991 analytic: 2.452991, relative error: 7.756009e-10
numerical: 1.437390 analytic: 1.437390, relative error: 4.783096e-10
numerical: -3.505081 analytic: -3.505081, relative error: 5.494732e-12
numerical: -0.306043 analytic: -0.306043, relative error: 1.483647e-08
numerical: -0.196161 analytic: -0.196161, relative error: 4.335526e-08
numerical: 0.493621 analytic: 0.493621, relative error: 9.948556e-09
numerical: 2.083216 analytic: 2.083216, relative error: 1.561584e-10
numerical: -0.455972 analytic: -0.455972, relative error: 2.562752e-09
numerical: 2.355970 analytic: 2.355970, relative error: 3.423371e-09
numerical: 0.062797 analytic: 0.062797, relative error: 3.783482e-08
numerical: -0.909063 analytic: -0.909063, relative error: 9.256917e-10
numerical: 0.598311 analytic: 0.598311, relative error: 6.752632e-09
numerical: -1.128044 analytic: -1.128044, relative error: 2.372706e-09
numerical: 1.851128 analytic: 1.851128, relative error: 2.213092e-09
numerical: -4.141493 analytic: -4.141493, relative error: 1.061986e-09
numerical: 1.511787 analytic: 1.511787, relative error: 6.103444e-09
numerical: 1.154629 analytic: 1.154629, relative error: 5.469700e-10
numerical: 1.903103 analytic: 1.903103, relative error: 2.986355e-09
numerical: 1.798967 analytic: 1.798967, relative error: 1.253227e-09
numerical: -2.926630 analytic: -2.926630, relative error: 2.896137e-09
numerical: 0.690422 analytic: 0.690422, relative error: 9.302278e-09
numerical: 2.240355 analytic: 2.240355, relative error: 1.987764e-09
```

```
numerical: 0.781490 analytic: 0.781490, relative error: 1.350538e-09
numerical: -0.078707 analytic: -0.078707, relative error: 5.274421e-08
numerical: -4.521821 analytic: -4.521821, relative error: 1.230528e-09
numerical: 3.534994 analytic: 3.534994, relative error: 1.196353e-09
numerical: -0.393301 analytic: -0.393301, relative error: 2.001020e-08
numerical: 0.230990 analytic: 0.230990, relative error: 1.103436e-08
numerical: -1.278633 analytic: -1.278633, relative error: 1.084142e-08
numerical: -1.857043 analytic: -1.857043, relative error: 4.397091e-10
numerical: -1.908500 analytic: -1.908500, relative error: 4.429745e-09
numerical: 2.023941 analytic: 2.023941, relative error: 1.799081e-09
numerical: 1.190224 analytic: 1.190224, relative error: 2.895509e-09
numerical: 1.931370 analytic: 1.931370, relative error: 1.387723e-10
numerical: -0.529996 analytic: -0.529996, relative error: 1.156999e-08
numerical: -3.369600 analytic: -3.369600, relative error: 2.140493e-10
numerical: 0.475437 analytic: 0.475437, relative error: 6.548477e-09
numerical: 1.858906 analytic: 1.858906, relative error: 5.194497e-10
numerical: -2.522404 analytic: -2.522404, relative error: 6.727589e-10
numerical: 2.051790 analytic: 2.051790, relative error: 1.503438e-09
numerical: 3.062723 analytic: 3.062723, relative error: 7.721220e-10
numerical: -0.161049 analytic: -0.161049, relative error: 1.760750e-08
numerical: 3.484952 analytic: 3.484952, relative error: 2.123174e-09
numerical: -5.387605 analytic: -5.387605, relative error: 2.797505e-10
numerical: -0.538544 analytic: -0.538544, relative error: 1.307919e-08
numerical: 0.161146 analytic: 0.161146, relative error: 4.421942e-08
numerical: 2.370896 analytic: 2.370896, relative error: 1.170576e-09
numerical: 0.505297 analytic: 0.505297, relative error: 1.353300e-08
numerical: -0.992979 analytic: -0.992979, relative error: 2.447346e-09
numerical: -2.500363 analytic: -2.500363, relative error: 3.497233e-10
numerical: 1.462460 analytic: 1.462460, relative error: 3.598319e-09
numerical: -2.577310 analytic: -2.577310, relative error: 3.125741e-09
numerical: -1.081324 analytic: -1.081324, relative error: 5.105599e-09
numerical: -0.358086 analytic: -0.358086, relative error: 4.319497e-08
numerical: 0.764854 analytic: 0.764854, relative error: 6.467525e-10
numerical: -3.535823 analytic: -3.535823, relative error: 2.415175e-09
numerical: 2.612689 analytic: 2.612689, relative error: 1.046387e-09
numerical: 1.878576 analytic: 1.878576, relative error: 1.629223e-10
numerical: -4.480881 analytic: -4.480881, relative error: 8.121022e-10
numerical: 2.657849 analytic: 2.657849, relative error: 2.408585e-11
numerical: -4.920479 analytic: -4.920479, relative error: 1.736727e-09
numerical: 0.686490 analytic: 0.686490, relative error: 1.209249e-08
numerical: -3.801261 analytic: -3.801261, relative error: 1.438660e-09
numerical: 0.145344 analytic: 0.145344, relative error: 6.999413e-09
numerical: 0.702848 analytic: 0.702848, relative error: 1.652842e-08
numerical: 0.892095 analytic: 0.892095, relative error: 8.438995e-09
numerical: 1.996837 analytic: 1.996837, relative error: 1.453998e-09
numerical: 1.493639 analytic: 1.493639, relative error: 2.795828e-10
numerical: 0.777351 analytic: 0.777351, relative error: 3.050778e-09
numerical: 1.137454 analytic: 1.137454, relative error: 5.727769e-09
```

```
numerical: -1.193187 analytic: -1.193187, relative error: 5.419247e-09
numerical: -3.550097 analytic: -3.550097, relative error: 1.961355e-09
numerical: 0.385890 analytic: 0.385890, relative error: 1.845966e-08
numerical: -0.484374 analytic: -0.484374, relative error: 7.656733e-09
numerical: 0.204579 analytic: 0.204579, relative error: 3.016325e-09
numerical: 0.790163 analytic: 0.790163, relative error: 4.785472e-09
numerical: 0.167805 analytic: 0.167805, relative error: 5.699567e-08
numerical: 2.208280 analytic: 2.208280, relative error: 2.434796e-09
numerical: -4.508730 analytic: -4.508730, relative error: 5.431798e-10
numerical: -2.153319 analytic: -2.153319, relative error: 1.178232e-09
numerical: 2.109852 analytic: 2.109852, relative error: 6.373476e-10
numerical: -4.652922 analytic: -4.652922, relative error: 4.706620e-10
numerical: -0.285630 analytic: -0.285630, relative error: 1.242358e-08
numerical: 0.615234 analytic: 0.615234, relative error: 5.964842e-09
numerical: 1.338863 analytic: 1.338863, relative error: 4.371954e-09
numerical: 0.830911 analytic: 0.830911, relative error: 5.019323e-09
numerical: 1.985753 analytic: 1.985753, relative error: 7.241509e-09
numerical: 3.535149 analytic: 3.535149, relative error: 9.700622e-10
numerical: 2.180266 analytic: 2.180266, relative error: 5.061417e-10
numerical: 1.350196 analytic: 1.350196, relative error: 9.929011e-10
numerical: 1.205004 analytic: 1.205004, relative error: 5.646401e-09
numerical: 2.865981 analytic: 2.865981, relative error: 8.561215e-10
numerical: 1.483164 analytic: 1.483164, relative error: 1.006305e-08
numerical: 1.046843 analytic: 1.046843, relative error: 3.885517e-09
numerical: 2.116976 analytic: 2.116976, relative error: 6.250997e-10
numerical: -0.708489 analytic: -0.708489, relative error: 6.004762e-09
numerical: 2.403440 analytic: 2.403441, relative error: 3.244190e-09
numerical: -0.234484 analytic: -0.234484, relative error: 3.289380e-08
numerical: 2.706114 analytic: 2.706114, relative error: 3.026567e-10
numerical: 2.345817 analytic: 2.345817, relative error: 9.605870e-10
numerical: 2.860311 analytic: 2.860311, relative error: 4.260679e-10
numerical: 0.756753 analytic: 0.756753, relative error: 4.794615e-09
numerical: -3.735726 analytic: -3.735726, relative error: 6.385231e-10
numerical: -0.085136 analytic: -0.085136, relative error: 1.104897e-07
numerical: 2.831937 analytic: 2.831937, relative error: 6.562682e-10
numerical: 2.294570 analytic: 2.294570, relative error: 1.962091e-10
numerical: 0.970525 analytic: 0.970525, relative error: 1.335590e-08
numerical: -0.969245 analytic: -0.969245, relative error: 2.812849e-09
numerical: -1.216701 analytic: -1.216701, relative error: 4.449819e-09
numerical: 0.900277 analytic: 0.900277, relative error: 3.948307e-09
numerical: -1.846592 analytic: -1.846592, relative error: 4.194752e-09
numerical: 0.159828 analytic: 0.159828, relative error: 2.984610e-08
numerical: 0.645183 analytic: 0.645183, relative error: 9.590565e-09
numerical: 2.341093 analytic: 2.341093, relative error: 2.586820e-09
numerical: -3.618277 analytic: -3.618277, relative error: 3.672384e-10
numerical: -0.936074 analytic: -0.936074, relative error: 3.005631e-10
numerical: 1.557078 analytic: 1.557078, relative error: 3.055674e-09
numerical: -0.414717 analytic: -0.414717, relative error: 1.183788e-08
```

```
numerical: 3.273538 analytic: 3.273538, relative error: 7.061635e-10
numerical: -0.360054 analytic: -0.360054, relative error: 2.817436e-08
numerical: -1.025533 analytic: -1.025533, relative error: 1.450865e-08
numerical: 0.049764 analytic: 0.049764, relative error: 8.466568e-08
numerical: 0.609717 analytic: 0.609717, relative error: 5.127375e-09
numerical: -4.110494 analytic: -4.110494, relative error: 2.348984e-09
numerical: 2.867318 analytic: 2.867318, relative error: 6.758203e-10
numerical: -1.970293 analytic: -1.970293, relative error: 3.878741e-09
numerical: 1.302603 analytic: 1.302603, relative error: 7.378249e-10
numerical: 1.524364 analytic: 1.524364, relative error: 3.601475e-09
numerical: 2.552454 analytic: 2.552454, relative error: 2.122618e-09
numerical: 0.976201 analytic: 0.976201, relative error: 4.684092e-10
numerical: -0.664603 analytic: -0.664603, relative error: 8.125714e-09
numerical: 2.779030 analytic: 2.779030, relative error: 2.864950e-09
numerical: 3.678296 analytic: 3.678296, relative error: 6.605413e-10
numerical: 1.156771 analytic: 1.156771, relative error: 1.365675e-09
numerical: 0.401995 analytic: 0.401995, relative error: 8.497192e-10
numerical: 1.405332 analytic: 1.405332, relative error: 6.644176e-11
numerical: -0.196066 analytic: -0.196066, relative error: 1.053366e-08
numerical: 0.992895 analytic: 0.992895, relative error: 8.694886e-09
numerical: -2.705058 analytic: -2.705058, relative error: 1.538586e-10
numerical: 0.509105 analytic: 0.509105, relative error: 3.074828e-09
numerical: 0.146351 analytic: 0.146351, relative error: 9.535272e-09
numerical: 2.793286 analytic: 2.793286, relative error: 3.144724e-09
numerical: -1.463958 analytic: -1.463958, relative error: 3.555088e-09
numerical: 2.421218 analytic: 2.421218, relative error: 2.746674e-09
numerical: -1.102373 analytic: -1.102373, relative error: 6.242229e-09
numerical: 0.059670 analytic: 0.059670, relative error: 9.000168e-08
numerical: 2.596098 analytic: 2.596098, relative error: 3.386918e-09
numerical: -4.963961 analytic: -4.963961, relative error: 4.210977e-10
numerical: 1.575603 analytic: 1.575603, relative error: 3.725412e-09
```

[5]:
```
# 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 cs682.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: %.28f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %.18f' % grad_difference)
```

```
naive loss: 2.390988e+00 computed in 0.083434s
vectorized loss: 2.390988e+00 computed in 0.093465s
Loss difference: 0.0000000000000004440892098501
Gradient difference: 0.000000000000369026
```

[6]:
```
# 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 cs682.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

################################################################################
# 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.                          ␣
 ↪#
################################################################################
regs = [2.4e4]
lrs = [2e-7]
for rg in regs:
    for lr in lrs:
        sfm = Softmax()
        loss_hist = sfm.train(X_train, y_train, learning_rate=lr, reg=rg,
                              num_iters=1500, verbose=True)
        y_train_pred = sfm.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = sfm.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)
        if best_val<val_acc:
```

```
            best_val=val_acc
            best_softmax=sfm
        results[(lr,rg)] = (train_acc,val_acc)
        print("reg: %d, lr %f, val_acc %f"%(rg,lr,val_acc))
################################################################################
#                               END OF YOUR CODE                               #
################################################################################

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

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

```
iteration 0 / 1500: loss 744.011142
iteration 100 / 1500: loss 109.067847
iteration 200 / 1500: loss 17.581898
iteration 300 / 1500: loss 4.363667
iteration 400 / 1500: loss 2.467312
iteration 500 / 1500: loss 2.125576
iteration 600 / 1500: loss 2.102611
iteration 700 / 1500: loss 2.107233
iteration 800 / 1500: loss 2.130462
iteration 900 / 1500: loss 2.083556
iteration 1000 / 1500: loss 2.042868
iteration 1100 / 1500: loss 2.068889
iteration 1200 / 1500: loss 2.067508
iteration 1300 / 1500: loss 2.067960
iteration 1400 / 1500: loss 2.138784
reg: 24000, lr 0.000000, val_acc 0.347000
lr 2.000000e-07 reg 2.400000e+04 train accuracy: 0.334102 val accuracy: 0.347000
best validation accuracy achieved during cross-validation: 0.347000
```

[7]:
```
# 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: 0.344000
```

Inline Question - *True or False*

It's 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 if we add a datapoint to the training set so that the loss value (si - sj) is negative for all values of i != j where j is the the correct class, then the net loss would still be the same. However in the case of Softmax classifier, the denominator sum is the sum of all score values (elements of the X.dot(W) matrix), hence it will change.

[8]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

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

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

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

two_layer_net

September 24, 2019

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

[1]:
```
# A bit of setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs682.classifiers.neural_net import TwoLayerNet


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

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

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cs682/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

[2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
```

```
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

2 Forward pass: compute scores

Open the file `cs682/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

[3]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
```

```
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720496109664e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

[4]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.794120407794253e-13
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

[5]:
```
from cs682.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward␣
 ↪pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
 ↪verbose=False)
```

```
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
↪grads[param_name])))
```

```
W1 max relative error: 3.669858e-09
W2 max relative error: 3.440708e-09
b1 max relative error: 2.738422e-09
b2 max relative error: 3.865028e-11
```

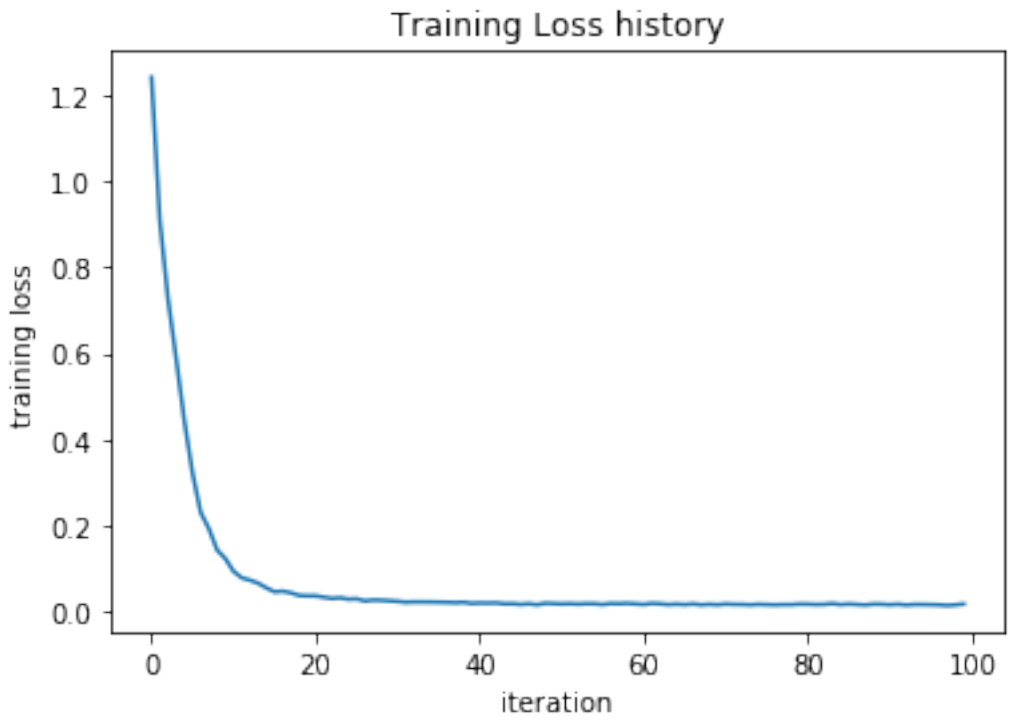
5 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

[6]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.017149607938732023
```



6 Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

[7]:
```
from cs682.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net 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 = 'cs682/datasets/cifar-10-batches-py'

    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]

    # 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

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

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


# 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

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = 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)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

7 Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

[8]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

8 Debug the training

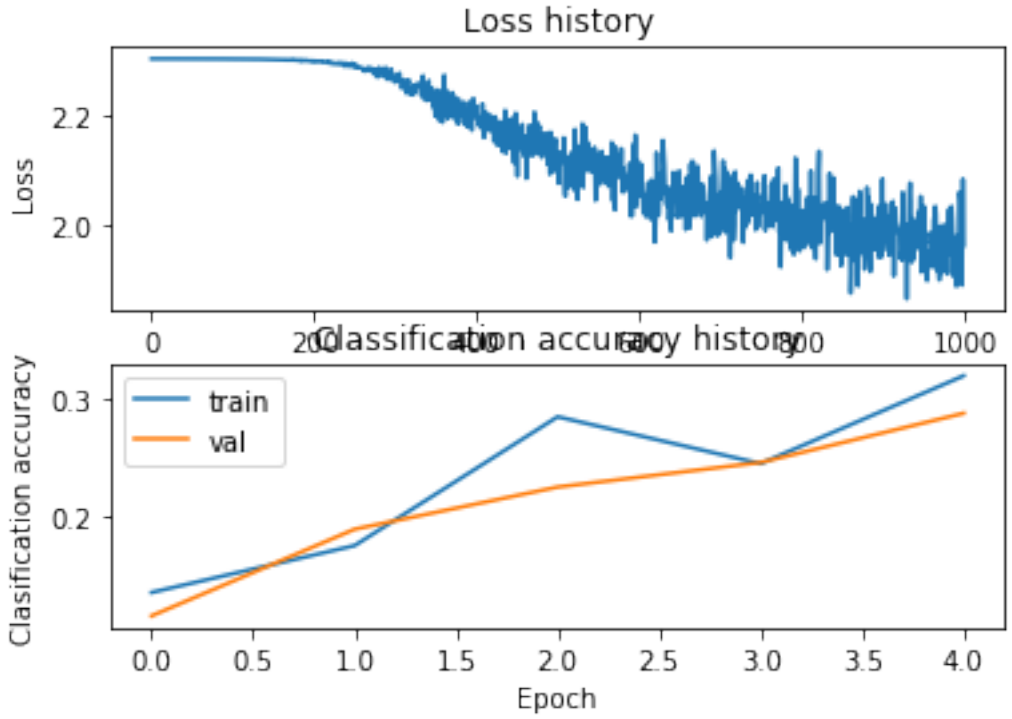
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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

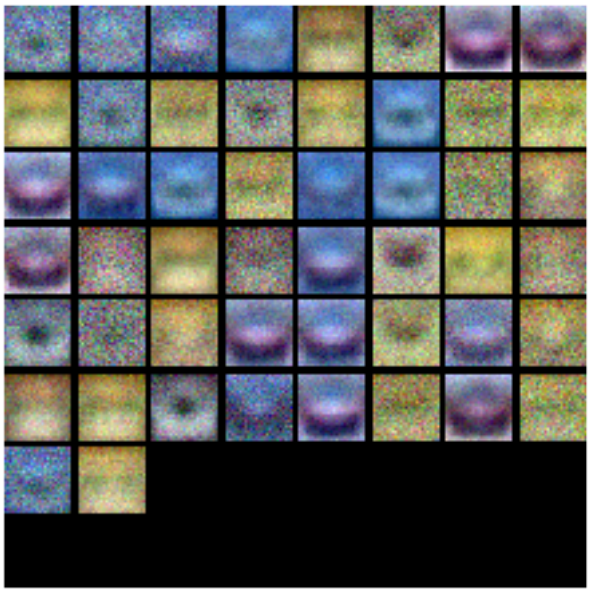
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[9]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')
```

```
plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```



[10]:
```
from cs682.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



9 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

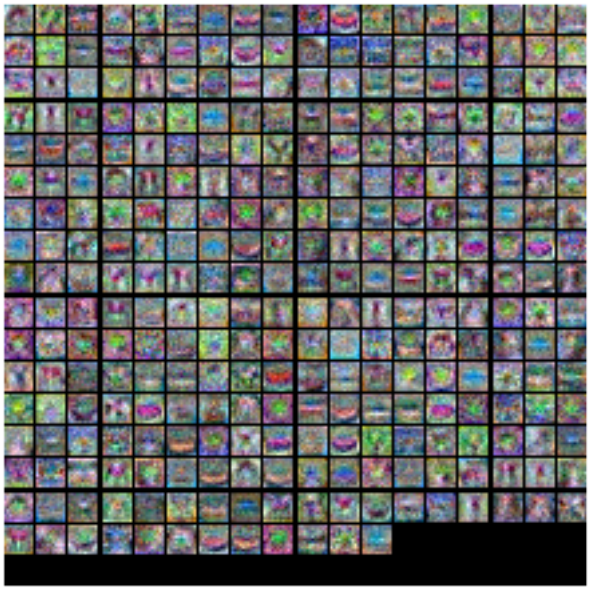
[11]:
```
best_net = None # store the best model into this

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained␣
 ↪#
# model in best_net.                                                           ␣
 ↪#
```

```
#                                                                               ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative    ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.          ␣
 ↪#
#                                                                               ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters          ␣
 ↪#
# automatically like we did on the previous exercises.                          ␣
 ↪#
################################################################################

best_val=-1
lrs = np.arange(4e-5,5e-5,1e-5).tolist()#[1e-7, 5e-5,1e-5]
regs = np.arange(0.25,1,0.25).tolist()#[0.25,1,0.25]
hs = np.arange(60,100,10).tolist()#[10,100,10]
batch_sizes = np.arange(200,1000,100).tolist()#[200,1000,100]
ni = np.arange(100,1000,100).tolist()

lrs = np.random.uniform(5e-6,5e-6,1).tolist()#[1e-7, 5e-5,1e-5]
regs = np.random.uniform(100,100,1).tolist()#[0.25,1,0.25]
hs = np.random.randint(60,61,1).tolist()#[10,100,10]
batch_sizes = np.random.randint(700,701,1).tolist()#[200,1000,100]

lrs=[1e-1,1e-2,1e-3,1e-4,1e-5,1e-6]
regs=[0.25,1,2,4,8,10]

lrs=[1e-3]
regs=[1e-2]
hs=[300]

for hidden_size in hs:
    for rg in regs:
        for lr in lrs:
            print("reg: %f, lr %f, hs %d"%(rg,lr,hidden_size))
            # Train the network
            net = TwoLayerNet(input_size, hidden_size, num_classes)
            stats = net.train(X_train, y_train, X_val, y_val,
                        num_iters=1000, batch_size=300,
                        learning_rate=lr, learning_rate_decay=0.95,
```

```
                        reg=rg, verbose=True)

            # Predict on the validation set
            val_acc = (net.predict(X_val) == y_val).mean()
            print('Validation accuracy: ', val_acc)
            if best_val<val_acc:
                best_val=val_acc
                best_net=net
################################################################################
#                               END OF YOUR CODE                               ␣
 ↪#
################################################################################
```

```
reg: 0.010000, lr 0.001000, hs 300
iteration 0 / 1000: loss 2.302667
iteration 100 / 1000: loss 1.872204
iteration 200 / 1000: loss 1.731257
iteration 300 / 1000: loss 1.607815
iteration 400 / 1000: loss 1.599375
iteration 500 / 1000: loss 1.497538
iteration 600 / 1000: loss 1.490515
iteration 700 / 1000: loss 1.382616
iteration 800 / 1000: loss 1.287533
iteration 900 / 1000: loss 1.338707
Validation accuracy:  0.491
```

[12]:
```
# visualize the weights of the best network
show_net_weights(best_net)
```



10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

[13]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.503
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply. 1. Train on a larger dataset. 2. Add more hidden units. 3. Increase the regularization strength. 4. None of the above.

Your answer: 1,3

Your explanation: 1. Training on a larger dataset would improve the model as statistically speaking it improves the probability of it scoring better for any test set. 2. Adding more hidden units can increase the training accuracy. By playing out with the examples it seems to be the case, it is able to accomodate more kind of data, however if we increase it beyond a range it will cause overfitting and not help in closing the gap between training and testing accuracy. 3. Regularization strength aims at avoiding the problem of overfitting to the training data, so increasing the regularisation strength may help in reducing the difference between training accuracy and testing accuracy, as our model would be not too specific.

features

September 24, 2019

1 Image features exercise

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

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

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

[1]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


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

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

1.1 Load data

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

[2]:
```
from cs682.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs682/datasets/cifar-10-batches-py'
```

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

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

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

# 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_val, y_val, X_test, y_test = get_CIFAR10_data()
```

1.2 Extract Features

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

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

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

[3]:
```
from cs682.features import *

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

```
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

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

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

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
```

```
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
```

1.3 Train SVM on features

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

[4]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs682.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-2]
regularization_strengths = [1e-3]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        ␣
 ↪#
# Use the validation set to set the learning rate and regularization strength.␣
 ↪#
# This should be identical to the validation that you did for the SVM; save    ␣
 ↪#
```

```
# the best trained classifer in best_svm. You might also want to play
# with different numbers of bins in the color histogram. If you are careful
# you should be able to get accuracy of near 0.44 on the validation set.
################################################################################

for rg in regularization_strengths:
    for i in range(len(learning_rates)):
        lr=learning_rates[i]
        print("reg: %d, lr %f"%(rg,lr))
        svm = LinearSVM()
        loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr, reg=rg,
                              num_iters=1000, verbose=True)
        y_train_pred = svm.predict(X_train_feats)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        val_acc = np.mean(y_val == y_val_pred)
        if best_val<val_acc:
            best_val=val_acc
            best_svm=svm
        results[(lr,rg)] = (train_acc,val_acc)
################################################################################
#                              END OF YOUR CODE
#
################################################################################

# 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)
```

```
reg: 0, lr 0.010000
iteration 0 / 1000: loss 9.004892
iteration 100 / 1000: loss 2.995509
iteration 200 / 1000: loss 3.432943
iteration 300 / 1000: loss 3.017958
iteration 400 / 1000: loss 3.535473
iteration 500 / 1000: loss 3.013970
iteration 600 / 1000: loss 3.101255
iteration 700 / 1000: loss 3.107034
iteration 800 / 1000: loss 3.176137
```

```
iteration 900 / 1000: loss 3.451010
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.509796 val accuracy: 0.499000
best validation accuracy achieved during cross-validation: 0.499000
```

[5]:
```python
# Evaluate your trained SVM on the test set
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.485
```

[6]:
```python
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

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

1.3.1 Inline question 1:

Describe the misclassification results that you see. Do they make sense? The misclassified results vaguely make sense as objects with the same shape are misclassed as one. Also if we try to increase the number of bins the pictures the misclassification with similar color increases.

1.4 Neural Network on image features

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

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

[7]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

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

[8]:
```
from cs682.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 300
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    ␣
 ↪#
# cross-validate various parameters as in previous sections. Store your best   ␣
 ↪#
# model in the best_net variable.                                              ␣
 ↪#
################################################################################
lrs=[0.8,1e-1,1e-2]
regs=[0.4e-7,2e-2,2e-8]
hs=[300]
best_net = None
best_val=-1
for hidden_size in hs:
    for rg in regs:
        for lr in lrs:
            print("reg: %f, lr %f, hs %d"%(rg,lr,hidden_size))
            # Train the network
            net = TwoLayerNet(input_dim, hidden_size, num_classes)
            stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                        num_iters=1000, batch_size=300,
                        learning_rate=lr, learning_rate_decay=0.95,
                        reg=rg, verbose=True)

            # Predict on the validation set
            val_acc = (net.predict(X_val_feats) == y_val).mean()
            print('Validation accuracy: ', val_acc)
            if best_val<val_acc:
                best_val=val_acc
                best_net=net

################################################################################
#                              END OF YOUR CODE                                ␣
 ↪#
################################################################################
```

```
reg: 0.000000, lr 0.800000, hs 300
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 1.379663
iteration 200 / 1000: loss 1.367033
```

```
iteration 300 / 1000: loss 1.243720
iteration 400 / 1000: loss 1.218183
iteration 500 / 1000: loss 1.149028
iteration 600 / 1000: loss 1.050437
iteration 700 / 1000: loss 0.981364
iteration 800 / 1000: loss 1.072331
iteration 900 / 1000: loss 1.026686
Validation accuracy:  0.546
reg: 0.000000, lr 0.100000, hs 300
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 2.302320
iteration 200 / 1000: loss 2.155152
iteration 300 / 1000: loss 1.889854
iteration 400 / 1000: loss 1.581867
iteration 500 / 1000: loss 1.535505
iteration 600 / 1000: loss 1.399941
iteration 700 / 1000: loss 1.369877
iteration 800 / 1000: loss 1.293121
iteration 900 / 1000: loss 1.373344
Validation accuracy:  0.517
reg: 0.000000, lr 0.010000, hs 300
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 2.302336
iteration 200 / 1000: loss 2.302355
iteration 300 / 1000: loss 2.302757
iteration 400 / 1000: loss 2.302517
iteration 500 / 1000: loss 2.302276
iteration 600 / 1000: loss 2.302542
iteration 700 / 1000: loss 2.302468
iteration 800 / 1000: loss 2.302918
iteration 900 / 1000: loss 2.302397
Validation accuracy:  0.078
reg: 0.020000, lr 0.800000, hs 300
iteration 0 / 1000: loss 2.302595
iteration 100 / 1000: loss 1.746846
iteration 200 / 1000: loss 1.746298
iteration 300 / 1000: loss 1.811395
iteration 400 / 1000: loss 1.804776
iteration 500 / 1000: loss 1.728867
iteration 600 / 1000: loss 1.771145
iteration 700 / 1000: loss 1.792670
iteration 800 / 1000: loss 1.711837
iteration 900 / 1000: loss 1.740108
Validation accuracy:  0.444
reg: 0.020000, lr 0.100000, hs 300
iteration 0 / 1000: loss 2.302595
iteration 100 / 1000: loss 2.303060
iteration 200 / 1000: loss 2.262500
```

```
iteration 300 / 1000: loss 2.034702
iteration 400 / 1000: loss 1.889555
iteration 500 / 1000: loss 1.858120
iteration 600 / 1000: loss 1.771601
iteration 700 / 1000: loss 1.787500
iteration 800 / 1000: loss 1.720810
iteration 900 / 1000: loss 1.669752
Validation accuracy:  0.472
reg: 0.020000, lr 0.010000, hs 300
iteration 0 / 1000: loss 2.302595
iteration 100 / 1000: loss 2.302675
iteration 200 / 1000: loss 2.302513
iteration 300 / 1000: loss 2.302532
iteration 400 / 1000: loss 2.302874
iteration 500 / 1000: loss 2.302462
iteration 600 / 1000: loss 2.302543
iteration 700 / 1000: loss 2.302440
iteration 800 / 1000: loss 2.302345
iteration 900 / 1000: loss 2.302418
Validation accuracy:  0.078
reg: 0.000000, lr 0.800000, hs 300
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 1.418549
iteration 200 / 1000: loss 1.325914
iteration 300 / 1000: loss 1.256266
iteration 400 / 1000: loss 1.165535
iteration 500 / 1000: loss 1.172471
iteration 600 / 1000: loss 1.062745
iteration 700 / 1000: loss 1.115724
iteration 800 / 1000: loss 0.902821
iteration 900 / 1000: loss 0.958637
Validation accuracy:  0.567
reg: 0.000000, lr 0.100000, hs 300
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 2.302575
iteration 200 / 1000: loss 2.157816
iteration 300 / 1000: loss 1.838520
iteration 400 / 1000: loss 1.694003
iteration 500 / 1000: loss 1.541360
iteration 600 / 1000: loss 1.442235
iteration 700 / 1000: loss 1.334927
iteration 800 / 1000: loss 1.320136
iteration 900 / 1000: loss 1.337232
Validation accuracy:  0.508
reg: 0.000000, lr 0.010000, hs 300
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 2.302575
iteration 200 / 1000: loss 2.302735
```

```
iteration 300 / 1000: loss 2.302531
iteration 400 / 1000: loss 2.302682
iteration 500 / 1000: loss 2.302456
iteration 600 / 1000: loss 2.302354
iteration 700 / 1000: loss 2.302239
iteration 800 / 1000: loss 2.302503
iteration 900 / 1000: loss 2.302298
Validation accuracy:  0.079
```

[9]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.568
```

[ ]: