knn

September 21, 2020

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

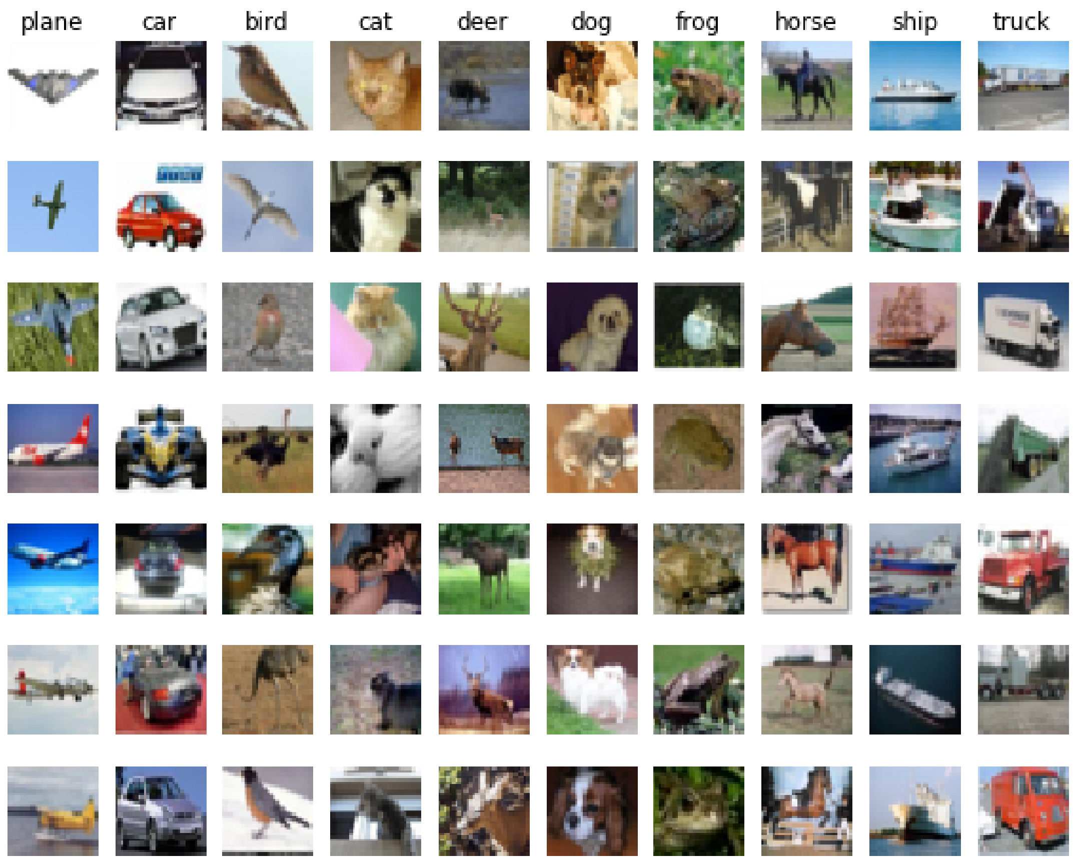
In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

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

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

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

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

[2]:
```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

[3]:
```python
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[4]:
```python
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

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

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

```
[5]: from cs231n.classifiers import KNearestNeighbor

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

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

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

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

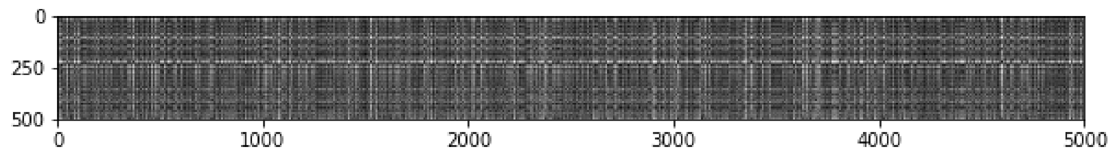
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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

```
[8]: # Open cs231n/classifiers/k_nearest_neighbor.py and implement
     # compute_distances_two_loops.

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

```
(500, 5000)
```

```
[9]: # 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?

$\color{blue}{Your Answer:}$ *fill this in.*

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

```
[4]
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`:

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

```
[4, 4, 4, 6, 6]
Got 139 / 500 correct => accuracy: 0.278000
```

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)}=p_{ij}^{(k)}-\mu$.)

2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean μ and dividing by the standard deviation σ. 4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}. 5. Rotating the coordinate axes of the data.

$Your Answer:$

$Your Explanation:$

[15]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

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

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

[18]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

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

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

[19]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took␣
↪to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

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

```
Two loop version took 1372.219490 seconds
One loop version took 41.698714 seconds
No loop version took 2.984686 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.

[32]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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

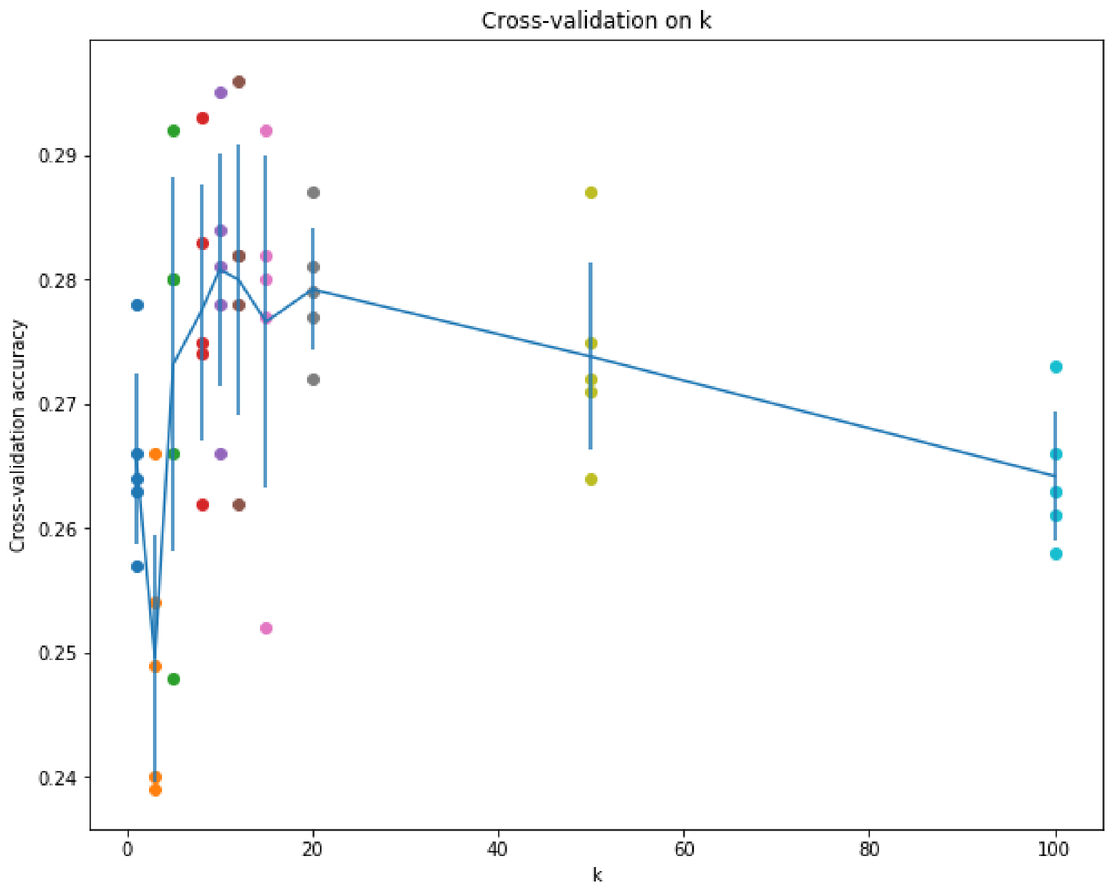
```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train, num_folds, axis = 0)
y_train_folds = np.array_split(y_train, num_folds, axis = 0)
# [print(a.shape) for a in X_train_folds]
pass

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

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



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

for k_trial in k_choices:
    acc = []
    for test_index in range(num_folds):
        XX_train = np.vstack(X_train_folds[:test_index] +␣
 ↪(X_train_folds[test_index+1:]
                                                            if test_index <␣
 ↪num_folds - 1 else []))
        yy_train = y_train_folds[:test_index] + (y_train_folds[test_index+1:]
                                                            if test_index <␣
 ↪num_folds - 1 else [])
        yy_train = np.hstack(yy_train)
        if k_trial == 1:
            print(XX_train.shape)
            print(yy_train.shape)

        XX_test = X_train_folds[test_index]
        yy_test = y_train_folds[test_index]

        classifier.train(XX_train, yy_train)
        y_test_pred = classifier.predict(XX_test, k=k_trial)
        acc = np.mean(y_test_pred == yy_test)
        k_to_accuracies.setdefault(k_trial, []).append(acc)

pass

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Print out the computed accuracies
print(k_to_accuracies)
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
(4000, 3072)
(4000,)
(4000, 3072)
(4000,)
(4000, 3072)
(4000,)
(4000, 3072)
(4000,)
(4000, 3072)
(4000,)
{1: [0.263, 0.257, 0.264, 0.278, 0.266], 3: [0.239, 0.249, 0.24, 0.266, 0.254],
5: [0.248, 0.266, 0.28, 0.292, 0.28], 8: [0.262, 0.283, 0.274, 0.293, 0.275],
10: [0.266, 0.295, 0.278, 0.284, 0.281], 12: [0.262, 0.296, 0.278, 0.282,
0.282], 15: [0.252, 0.292, 0.28, 0.282, 0.277], 20: [0.272, 0.281, 0.277, 0.279,
0.287], 50: [0.272, 0.287, 0.275, 0.264, 0.271], 100: [0.261, 0.273, 0.263,
0.258, 0.266]}
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 = 0.283000
k = 8, accuracy = 0.274000
k = 8, accuracy = 0.293000
```

```
k = 8, accuracy = 0.275000
k = 10, accuracy = 0.266000
k = 10, accuracy = 0.295000
k = 10, accuracy = 0.278000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.281000
k = 12, accuracy = 0.262000
k = 12, accuracy = 0.296000
k = 12, accuracy = 0.278000
k = 12, accuracy = 0.282000
k = 12, accuracy = 0.282000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.280000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.277000
k = 20, accuracy = 0.272000
k = 20, accuracy = 0.281000
k = 20, accuracy = 0.277000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.287000
k = 50, accuracy = 0.272000
k = 50, accuracy = 0.287000
k = 50, accuracy = 0.275000
k = 50, accuracy = 0.264000
k = 50, accuracy = 0.271000
k = 100, accuracy = 0.261000
k = 100, accuracy = 0.273000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.258000
k = 100, accuracy = 0.266000
```

[33]:

```
# 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()
```



[34]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

```
Got 140 / 500 correct => accuracy: 0.280000
```

Inline Question 3

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

$Your Answer:$

2,4

$Your Explanation:$

only explanate for 2: to calculate the train error, the k-NN will find the k nearest sample in the train dataset and vote for the class. In the case k=1, the nearest sample is itself, so it is always correct for train dataset.

svm

September 21, 2020

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

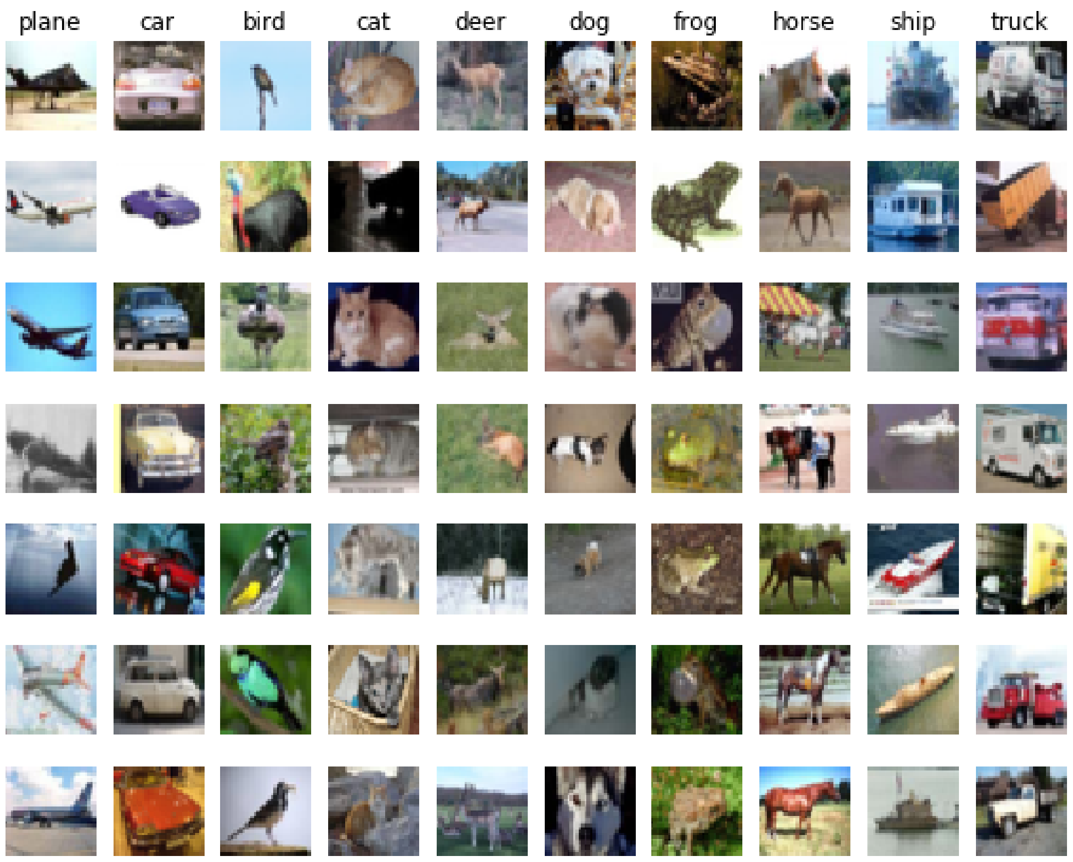
[2]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

[3]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

[4]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[5]:
```
# 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('Development data shape: ', X_dev.shape)
print('Development labels shape: ', y_dev.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

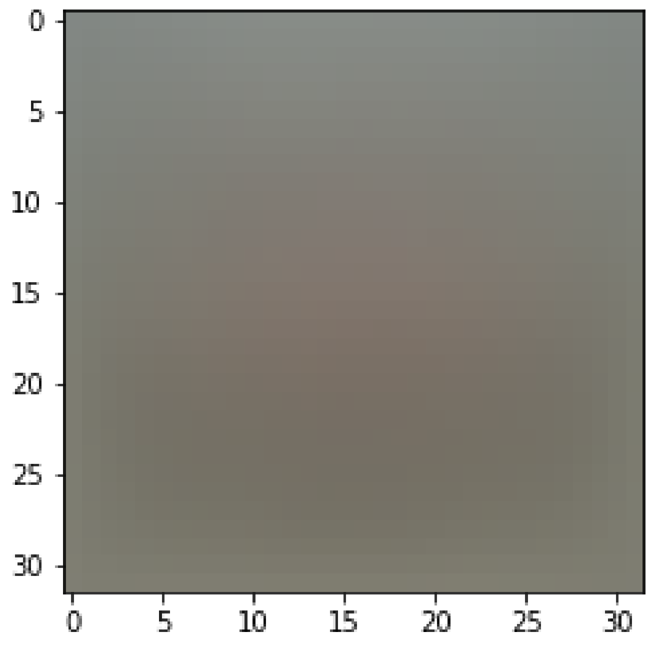
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Development data shape:  (500, 32, 32, 3)
Development labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

[7]:
```python
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
 ↪image
plt.show()

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

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

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

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



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

1.2 SVM Classifier

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

As you can see, we have prefilled the function `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

[8]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 8.448921
```

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:

[9]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

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

```
numerical: -16.119539 analytic: -16.119539, relative error: 2.178458e-11
numerical: 14.790645 analytic: 14.790645, relative error: 5.071104e-12
numerical: 17.405576 analytic: 17.405576, relative error: 1.016914e-11
numerical: 14.272671 analytic: 14.272671, relative error: 8.399090e-13
numerical: 26.322872 analytic: 26.322872, relative error: 4.401470e-12
numerical: -5.360679 analytic: -5.360679, relative error: 3.559165e-11
numerical: -9.977080 analytic: -9.977080, relative error: 1.234805e-11
numerical: 17.887227 analytic: 17.887227, relative error: 1.439976e-14
numerical: 24.688428 analytic: 24.688428, relative error: 4.643933e-12
numerical: -30.311840 analytic: -30.311840, relative error: 1.094353e-11
numerical: -44.479039 analytic: -44.479039, relative error: 7.868536e-12
numerical: 9.363879 analytic: 9.363879, relative error: 2.300416e-11
numerical: -24.412533 analytic: -24.412533, relative error: 1.325617e-12
numerical: -10.140507 analytic: -10.095886, relative error: 2.205011e-03
numerical: -28.473264 analytic: -28.473264, relative error: 2.784075e-12
numerical: 3.623001 analytic: 3.623001, relative error: 9.784382e-11
numerical: 7.454451 analytic: 7.504446, relative error: 3.342170e-03
numerical: -21.472794 analytic: -21.472794, relative error: 2.401087e-11
numerical: 7.541492 analytic: 7.541492, relative error: 1.701997e-11
numerical: 5.283373 analytic: 5.283373, relative error: 1.793941e-11
```

Inline Question 1

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

$Your Answer:$ *fill this in.*

[10]:
```
# Next implement the function svm_loss_vectorized; for now only compute the␣
 ↪loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print(loss_vectorized)
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much␣
 ↪faster.
```

```
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.448921e+00 computed in 0.438213s
8.448920718930303
Vectorized loss: 8.448921e+00 computed in 0.009994s
difference: -0.000000
```

[17]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

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

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

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

```
Naive loss and gradient: computed in 0.245330s
(500, 10)
Vectorized loss and gradient: computed in 0.008282s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

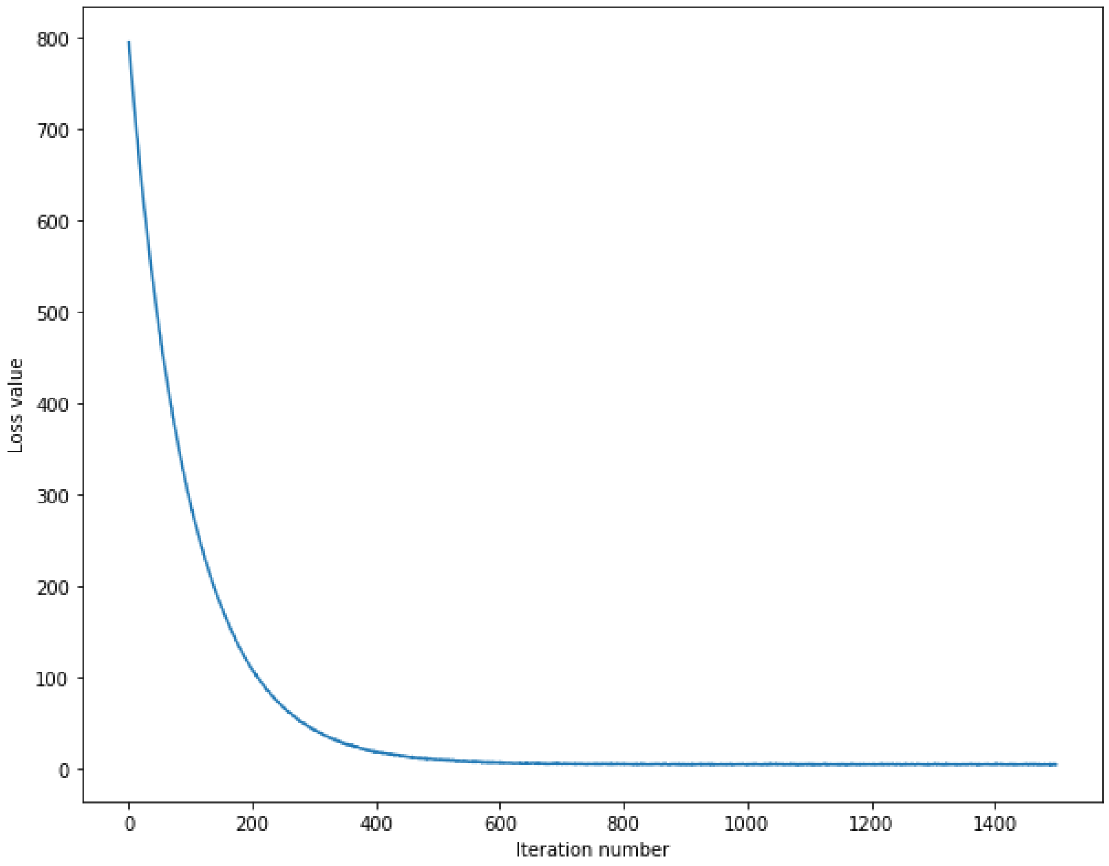
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[20]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
```

```
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 794.581734
iteration 100 / 1500: loss 289.325799
iteration 200 / 1500: loss 108.371408
iteration 300 / 1500: loss 43.303534
iteration 400 / 1500: loss 18.982181
iteration 500 / 1500: loss 10.209466
iteration 600 / 1500: loss 7.035879
iteration 700 / 1500: loss 5.698621
iteration 800 / 1500: loss 5.519656
iteration 900 / 1500: loss 4.724307
iteration 1000 / 1500: loss 5.668572
iteration 1100 / 1500: loss 5.582209
iteration 1200 / 1500: loss 5.168830
iteration 1300 / 1500: loss 4.998962
iteration 1400 / 1500: loss 5.223900
That took 7.067567s
```

[21]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



[22]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.372327
validation accuracy: 0.376000
```

[23]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

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

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

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

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

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

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

# Print out results.
```

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

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

```
iteration 0 / 1500: loss 785.539266
iteration 100 / 1500: loss 286.692953
iteration 200 / 1500: loss 108.118049
iteration 300 / 1500: loss 42.420597
iteration 400 / 1500: loss 18.494778
iteration 500 / 1500: loss 10.905700
iteration 600 / 1500: loss 6.759131
iteration 700 / 1500: loss 6.126146
iteration 800 / 1500: loss 5.653723
iteration 900 / 1500: loss 5.070240
iteration 1000 / 1500: loss 5.218781
iteration 1100 / 1500: loss 5.485191
iteration 1200 / 1500: loss 5.088635
iteration 1300 / 1500: loss 5.866981
iteration 1400 / 1500: loss 5.551097
iteration 0 / 1500: loss 1567.535291
iteration 100 / 1500: loss 210.838526
iteration 200 / 1500: loss 32.754900
iteration 300 / 1500: loss 9.024653
iteration 400 / 1500: loss 6.287248
iteration 500 / 1500: loss 5.931232
iteration 600 / 1500: loss 5.883757
iteration 700 / 1500: loss 5.323514
iteration 800 / 1500: loss 5.656877
iteration 900 / 1500: loss 5.818975
iteration 1000 / 1500: loss 5.628977
iteration 1100 / 1500: loss 6.090968
iteration 1200 / 1500: loss 5.670273
iteration 1300 / 1500: loss 5.449821
iteration 1400 / 1500: loss 6.012988
iteration 0 / 1500: loss 798.740448
iteration 100 / 1500: loss 397501181051904790757143901749735587840.000000
iteration 200 / 1500: loss 6570376167225323834887463695629688187651958998695673
4
664450254665068576768.000000
iteration 300 / 1500: loss 1086030558817522412188700103803385538922772594260869
2
974502928220571625454036267054006260146123062362071105536.000000
iteration 400 / 1500: loss 1795121534393955456034367162403176026451259643064169
9
227666271649512469380328213135621853967927869069155228503132577416171422902504
29
381455380480.000000
```

```
iteration 500 / 1500: loss 29671921264844952322776202722703115584269237491239292
19744430877443779040803213003920189544782410658945117268035152286564246211626354
6333400626940353825588683849231602172114344345 6.000000
iteration 600 / 1500: loss 49045309450003030663908574113209708342766354320472826
96854738349583871108006720139401120797136134695667264974876769935382300760834481
46455562069711716155488025412480709140626367640774484384464338962860881287294484
48.000000
iteration 700 / 1500: loss 81067968520680344212369626071236858720321960795184546
38066269055677368075296178426238838632043185843111338254031185330743122960411450
77648065931490468940126555287251654396819351241621456400222469977019053998030286
101388304047828989825369902328211046 4.000000
iteration 800 / 1500: loss 13399885929498638910456333500942284020809264698239717
94643992490069562116039101438017774014081498821994652451718738855632068051947810
94840382324109506741514554066707022971460259213255160295829930496547763224284297
19142134304841542325048645636063460597995005324054199884241232692594606 08.000000
```

```
/home/yaojian/project/ml_project/cnn_cs231n/assignment1/assignment1_jupyter/cs23
1n/classifiers/linear_svm.py:93: RuntimeWarning: overflow encountered in
double_scalars
  loss += reg * np.sum(W * W)
/home/yaojian/miniconda3/envs/cs231n/lib/python3.7/site-
packages/numpy/core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/yaojian/project/ml_project/cnn_cs231n/assignment1/assignment1_jupyter/cs23
1n/classifiers/linear_svm.py:93: RuntimeWarning: overflow encountered in
multiply
  loss += reg * np.sum(W * W)
```

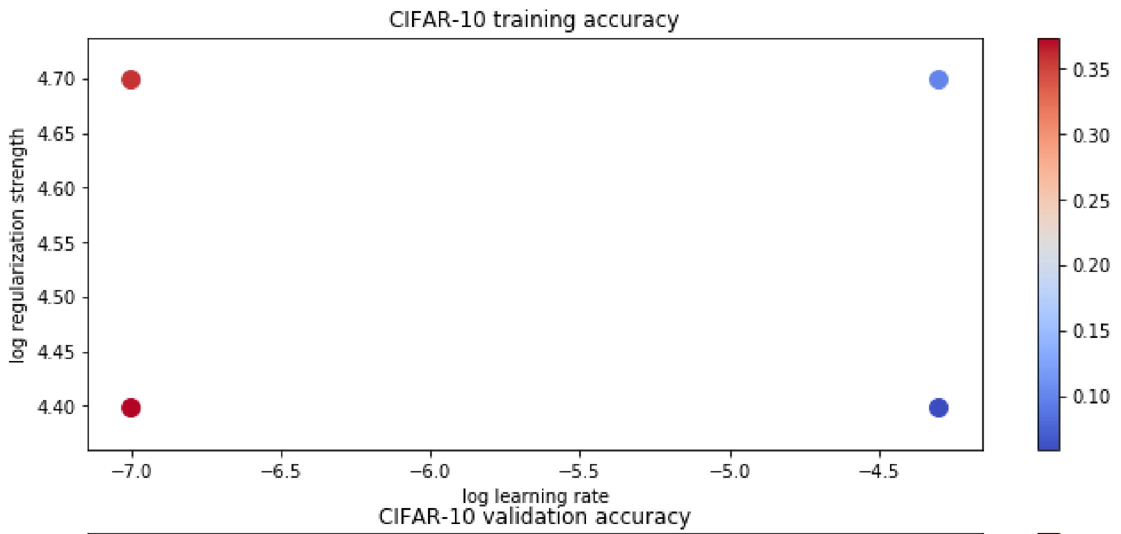
```
iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
iteration 0 / 1500: loss 1543.996765
iteration 100 / 1500: loss 42778722824930124895716295124615096863321660001221460
5622443131931291445172407579497884776410502690765826448155331348398080 0.000000
iteration 200 / 1500: loss 11046535179938946883848432144110431664244724141465172
64743317041184307755272220137496289434343668334103821108399208535878519752196939
39680925689032371658296616140116061187607242651109746173338027079586650641557792
04303310526305566264730111180800.000000
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf
```

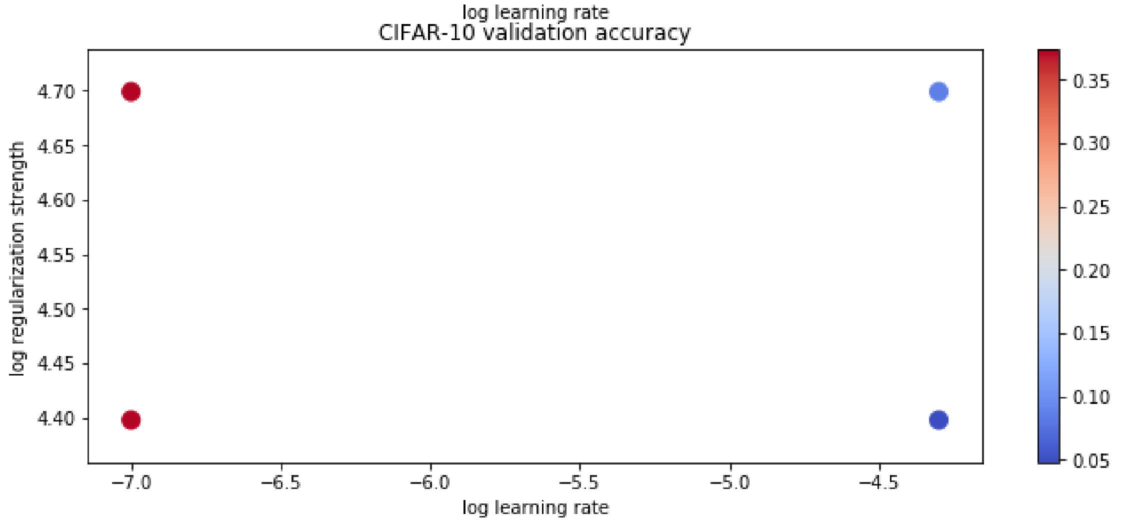
```
/home/yaojian/project/ml_project/cnn_cs231n/assignment1/assignment1_jupyter/cs23
1n/classifiers/linear_svm.py:118: RuntimeWarning: overflow encountered in
multiply
```

```
  dW = X.transpose().dot(mat_to_sum)/num_train
/home/yaojian/project/ml_project/cnn_cs231n/assignment1/assignment1_jupyter/cs23
1n/classifiers/linear_svm.py:112: RuntimeWarning: invalid value encountered in
greater
  iterm4 = (iterm1 - iterm2 + 1 > 0).astype(float)
/home/yaojian/project/ml_project/cnn_cs231n/assignment1/assignment1_jupyter/cs23
1n/classifiers/linear_classifier.py:76: RuntimeWarning: invalid value
encountered in subtract
  self.W -= learning_rate * grad
```

```
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.373898 val accuracy: 0.374000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.358959 val accuracy: 0.374000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.058653 val accuracy: 0.048000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.374000
```

[25]:
```
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

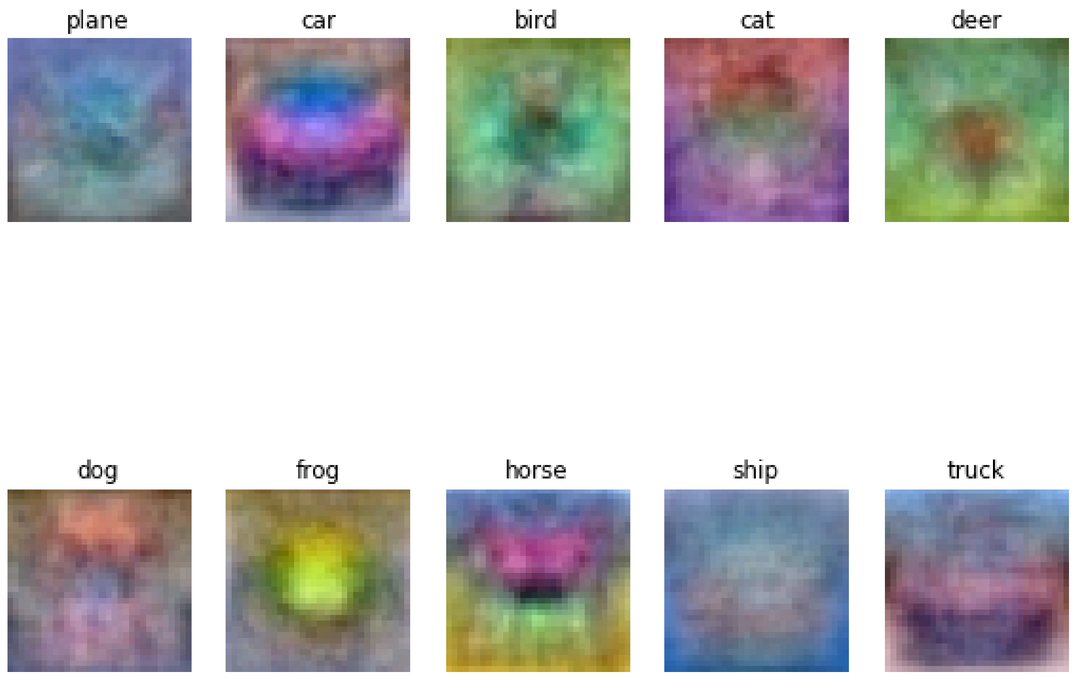




[26]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.376000
```

[27]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



Inline question 2

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

Your Answer : *skip*

```
[ ]:
```

two_layer_net

September 21, 2020

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.

[49]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.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))))
```

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

We will use the class `TwoLayerNet` in the file `cs231n/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.

[50]:
```
# 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 `cs231n/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.

[51]:
```
print("X: ", X, "\n", "y: ", y)
```

```
X:  [[ 16.24345364  -6.11756414  -5.28171752 -10.72968622]
 [  8.65407629 -23.01538697  17.44811764  -7.61206901]
 [  3.19039096  -2.49370375  14.62107937 -20.60140709]
 [ -3.22417204  -3.84054355  11.33769442 -10.99891267]
 [ -1.72428208  -8.77858418   0.42213747   5.82815214]]
 y:  [0 1 2 2 1]
```

[52]:
```
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:
3.6802720745909845e-08
```

3 Forward pass: compute loss

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

[53]:
```
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:
1.7985612998927536e-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:

[54]:
```
from cs231n.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])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447646e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

[36]:
```
np.array([[1,2,3],[2,3,3]])/np.array([2,3,2])
```

[36]:
```
array([[0.5       , 0.66666667, 1.5       ],
       [1.        , 1.        , 1.5       ]])
```

5 Train the network

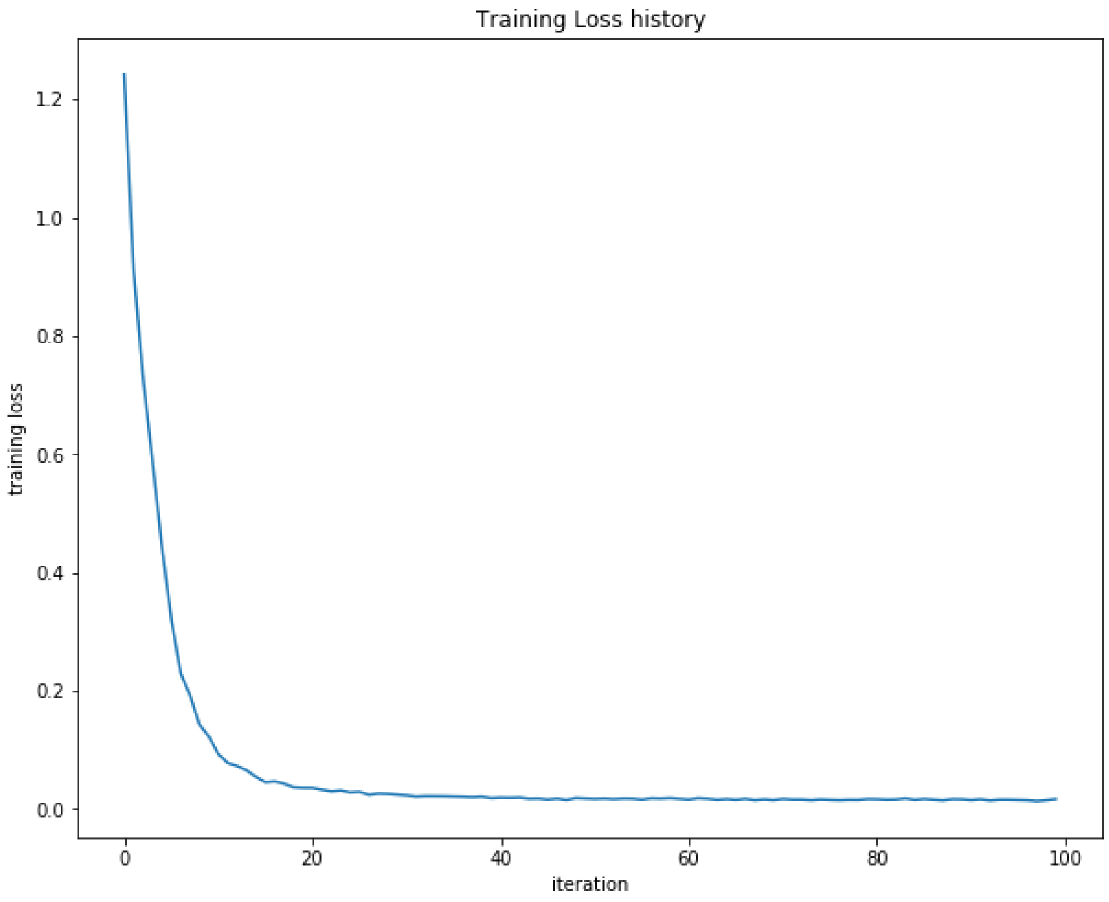
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and 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 0.02.

[59]:
```
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:  0.017149612521020405
```



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.

[60]:
```
from cs231n.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 = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

    # 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


# 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.

[61]:
```
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
```

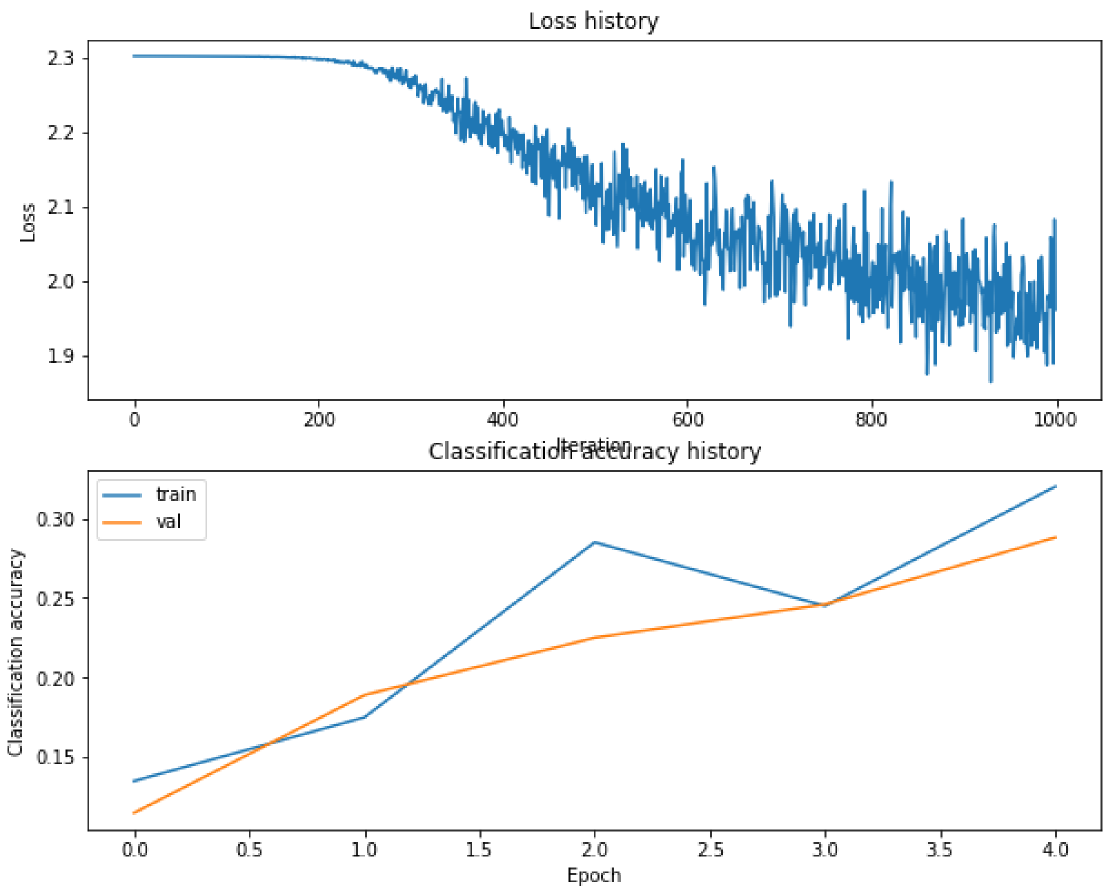
8 Debug the training

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.

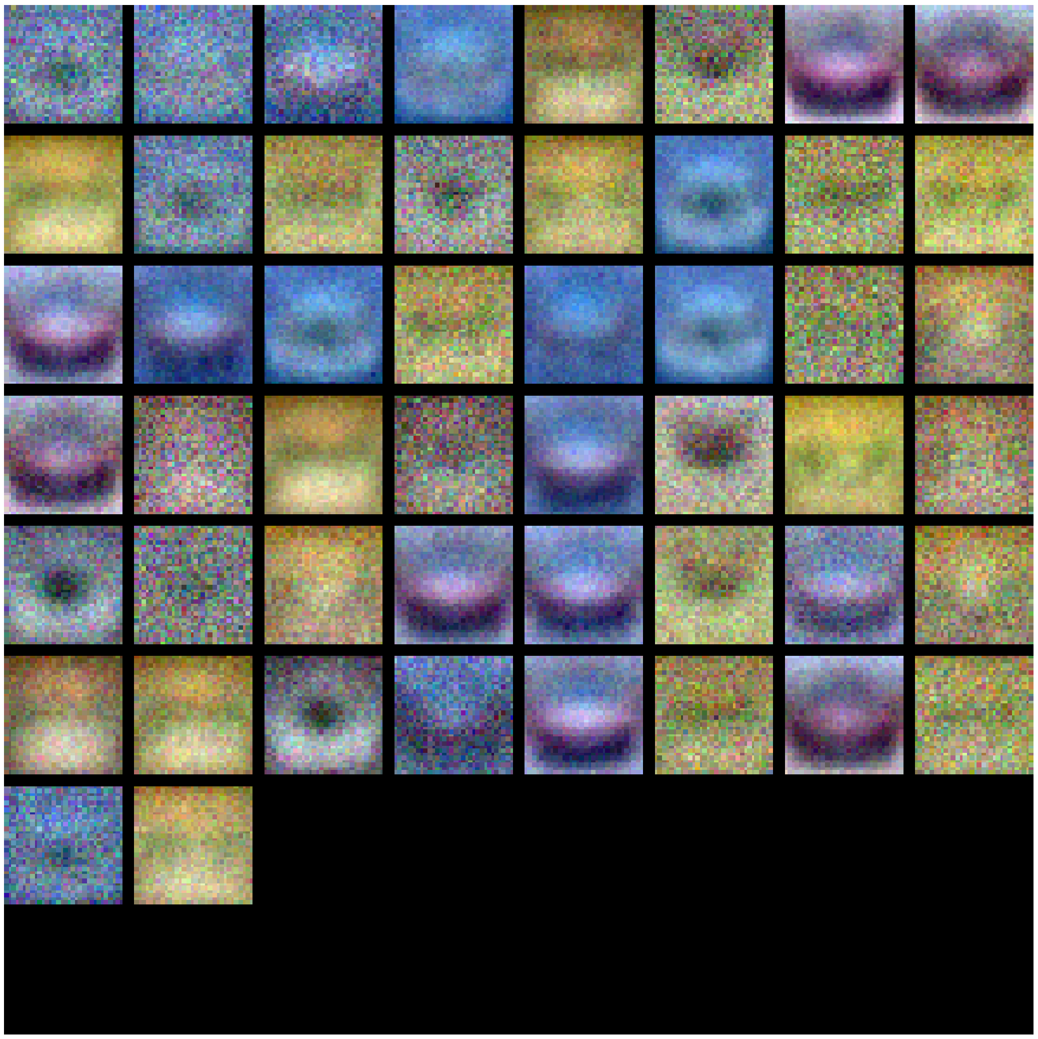
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[62]:
```
# 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('Classification accuracy')
plt.legend()
plt.show()
```



```
[63]: from cs231n.vis_utils import visualize_grid

      # Visualize the weights of the network

      def show_net_weights(net):
          W1 = net.params['W1']
          W1 = W1.reshape(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 (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

Explain your hyperparameter tuning process below.

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

[73]:
```
best_net = None # store the best model into this
best_val = 0
results = {}
#################################################################################
# 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.                        ␣
 ↪#
#################################################################################


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

learning_rates = [1e-3, 2.5e-4, 5e-4]
regularization_strengths = [0.25, 0.50]

input_size = 32 * 32 * 3
```

```
hidden_size = 1024
num_classes = 10

for lr in learning_rates:
    for reg in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, num_classes)
        net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=lr, learning_rate_decay=0.95,
            reg=reg, verbose=True)
        y_val_pred = net.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_net = net

        y_train_pred = net.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        results[(lr, reg)] = (train_accuracy, val_accuracy)
pass

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

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

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

```
iteration 0 / 1000: loss 2.310470
iteration 100 / 1000: loss 1.984113
iteration 200 / 1000: loss 1.743323
iteration 300 / 1000: loss 1.731493
iteration 400 / 1000: loss 1.640278
iteration 500 / 1000: loss 1.601560
iteration 600 / 1000: loss 1.596827
iteration 700 / 1000: loss 1.565236
iteration 800 / 1000: loss 1.443891
iteration 900 / 1000: loss 1.377435
iteration 0 / 1000: loss 2.318305
iteration 100 / 1000: loss 1.857280
iteration 200 / 1000: loss 1.716086
iteration 300 / 1000: loss 1.571621
iteration 400 / 1000: loss 1.657317
iteration 500 / 1000: loss 1.689210
iteration 600 / 1000: loss 1.668167
iteration 700 / 1000: loss 1.578514
iteration 800 / 1000: loss 1.560481
iteration 900 / 1000: loss 1.621917
iteration 0 / 1000: loss 2.310486
iteration 100 / 1000: loss 2.221606
iteration 200 / 1000: loss 2.011748
iteration 300 / 1000: loss 1.915786
iteration 400 / 1000: loss 1.820383
iteration 500 / 1000: loss 1.860659
iteration 600 / 1000: loss 1.744002
iteration 700 / 1000: loss 1.608366
iteration 800 / 1000: loss 1.739014
iteration 900 / 1000: loss 1.690952
iteration 0 / 1000: loss 2.318343
iteration 100 / 1000: loss 2.219664
iteration 200 / 1000: loss 2.066094
iteration 300 / 1000: loss 1.912767
iteration 400 / 1000: loss 1.968429
iteration 500 / 1000: loss 1.815977
iteration 600 / 1000: loss 1.856186
iteration 700 / 1000: loss 1.783355
iteration 800 / 1000: loss 1.771749
iteration 900 / 1000: loss 1.726225
iteration 0 / 1000: loss 2.310321
iteration 100 / 1000: loss 2.023392
iteration 200 / 1000: loss 1.730667
iteration 300 / 1000: loss 1.762334
iteration 400 / 1000: loss 1.785327
iteration 500 / 1000: loss 1.660141
iteration 600 / 1000: loss 1.696189
iteration 700 / 1000: loss 1.590018
iteration 800 / 1000: loss 1.632331
iteration 900 / 1000: loss 1.474059
iteration 0 / 1000: loss 2.318439
iteration 100 / 1000: loss 2.065190
iteration 200 / 1000: loss 1.894954
iteration 300 / 1000: loss 1.850756
iteration 400 / 1000: loss 1.783379
iteration 500 / 1000: loss 1.645474
iteration 600 / 1000: loss 1.776571
iteration 700 / 1000: loss 1.639906
iteration 800 / 1000: loss 1.578897
iteration 900 / 1000: loss 1.719972
lr 2.500000e-04 reg 2.500000e-01 train accuracy: 0.414347 val accuracy: 0.432000
lr 2.500000e-04 reg 5.000000e-01 train accuracy: 0.411796 val accuracy: 0.411000
lr 5.000000e-04 reg 2.500000e-01 train accuracy: 0.472327 val accuracy: 0.471000
lr 5.000000e-04 reg 5.000000e-01 train accuracy: 0.460653 val accuracy: 0.455000
lr 1.000000e-03 reg 2.500000e-01 train accuracy: 0.520673 val accuracy: 0.497000
lr 1.000000e-03 reg 5.000000e-01 train accuracy: 0.501061 val accuracy: 0.467000
best validation accuracy achieved during cross-validation: 0.497000
```

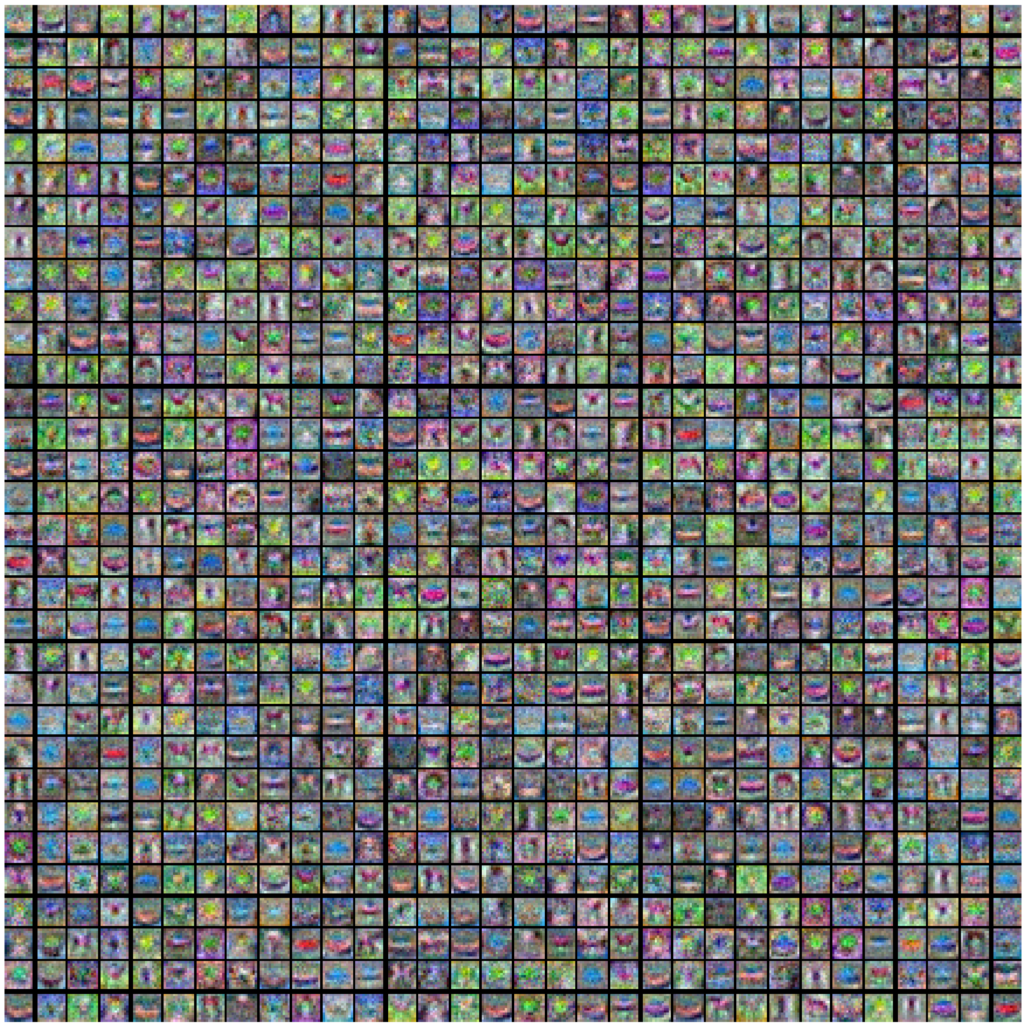
10 Note added by writer !!!

This example tells us that the when the loss is decreasing more or less linearly, it suggests 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.

[70]:
```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.489
```

[71]:
```
# Visualize the weights of the best network
show_net_weights(best_net)
```



11 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%.

[72]:
```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.488
```

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 :

features

September 21, 2020

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


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

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

1.1 Load data

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

[2]:
```
from cs231n.features import color_histogram_hsv, hog_feature

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

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

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

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

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

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

1.2 Extract Features

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

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

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

[3]:
```
from cs231n.features import *

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

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

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

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

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

1.3 Train SVM on features

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

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

```python
from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 1e-6]
regularization_strengths = [2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# TODO: can be done in parallel -------------------->
for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,
                  num_iters=1500, verbose=True)
        y_val_pred = svm.predict(X_val_feats)
        acc_val = np.mean(y_val == y_val_pred)
        if acc_val > best_val:
            best_val = acc_val
            best_svm = svm
        y_train_pred = svm.predict(X_train_feats)
        acc_train = np.mean(y_train == y_train_pred)
        results[(lr, reg)] = (acc_train, acc_val)
pass

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

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

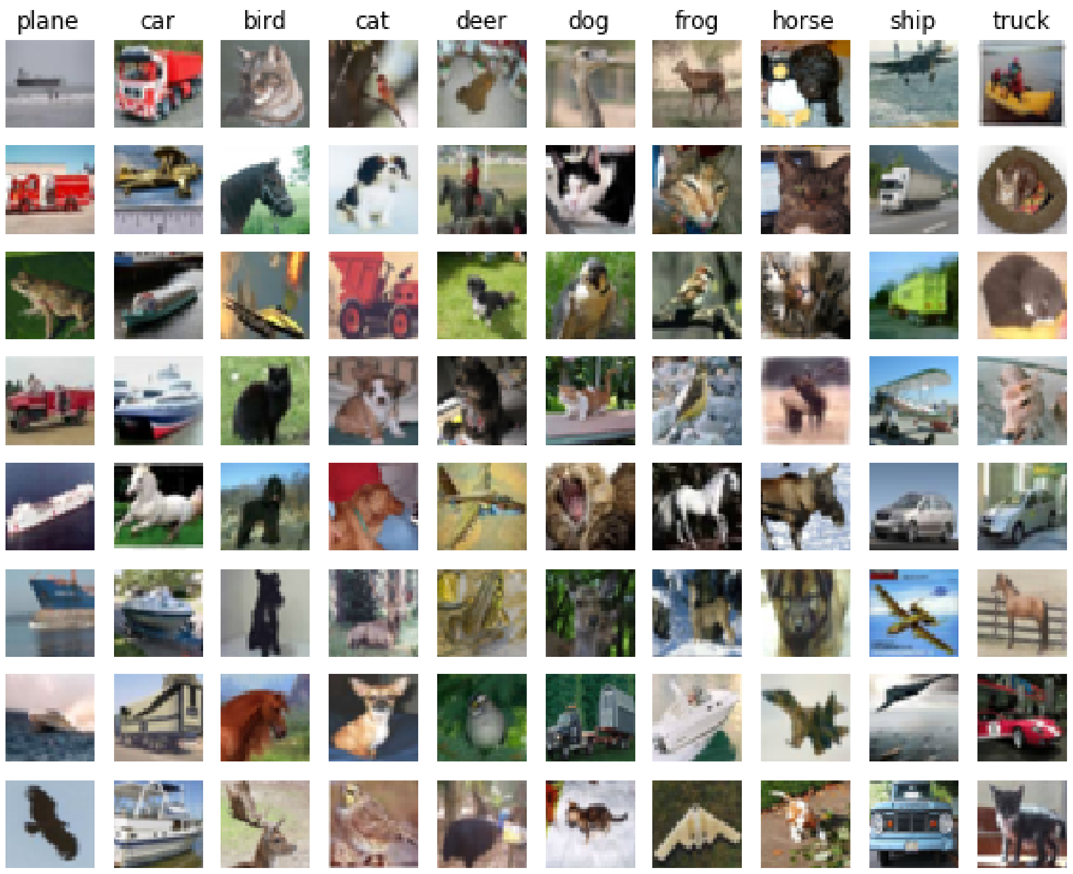
```
print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
iteration 0 / 1500: loss 49.812945
iteration 100 / 1500: loss 23.976447
iteration 200 / 1500: loss 14.494139
iteration 300 / 1500: loss 11.013618
iteration 400 / 1500: loss 9.738179
iteration 500 / 1500: loss 9.271493
iteration 600 / 1500: loss 9.099428
iteration 700 / 1500: loss 9.035894
iteration 800 / 1500: loss 9.012955
iteration 900 / 1500: loss 9.004251
iteration 1000 / 1500: loss 9.001103
iteration 1100 / 1500: loss 8.999919
iteration 1200 / 1500: loss 8.999620
iteration 1300 / 1500: loss 8.999500
iteration 1400 / 1500: loss 8.999308
iteration 1499 / 1500: loss 8.999316
iteration 0 / 1500: loss 85.815486
iteration 100 / 1500: loss 19.290904
iteration 200 / 1500: loss 10.377452
iteration 300 / 1500: loss 9.184651
iteration 400 / 1500: loss 9.024219
iteration 500 / 1500: loss 9.002905
iteration 600 / 1500: loss 9.000160
iteration 700 / 1500: loss 8.999713
iteration 800 / 1500: loss 8.999723
iteration 900 / 1500: loss 8.999664
iteration 1000 / 1500: loss 8.999558
iteration 1100 / 1500: loss 8.999705
iteration 1200 / 1500: loss 8.999684
iteration 1300 / 1500: loss 8.999716
iteration 1400 / 1500: loss 8.999659
iteration 1499 / 1500: loss 8.999600
iteration 0 / 1500: loss 47.435288
iteration 100 / 1500: loss 9.000638
iteration 200 / 1500: loss 8.999398
iteration 300 / 1500: loss 8.999295
iteration 400 / 1500: loss 8.999307
iteration 500 / 1500: loss 8.999137
iteration 600 / 1500: loss 8.999263
iteration 700 / 1500: loss 8.999318
iteration 800 / 1500: loss 8.999413
iteration 900 / 1500: loss 8.999367
iteration 1000 / 1500: loss 8.999248
iteration 1100 / 1500: loss 8.999488
iteration 1200 / 1500: loss 8.999364
iteration 1300 / 1500: loss 8.999341
iteration 1400 / 1500: loss 8.999329
iteration 1499 / 1500: loss 8.999290
iteration 0 / 1500: loss 84.417451
iteration 100 / 1500: loss 8.999597
iteration 200 / 1500: loss 8.999654
iteration 300 / 1500: loss 8.999663
iteration 400 / 1500: loss 8.999625
iteration 500 / 1500: loss 8.999632
iteration 600 / 1500: loss 8.999679
iteration 700 / 1500: loss 8.999717
iteration 800 / 1500: loss 8.999662
iteration 900 / 1500: loss 8.999705
iteration 1000 / 1500: loss 8.999723
iteration 1100 / 1500: loss 8.999645
iteration 1200 / 1500: loss 8.999755
iteration 1300 / 1500: loss 8.999662
iteration 1400 / 1500: loss 8.999663
iteration 1499 / 1500: loss 8.999662
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.414878 val accuracy: 0.421000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.412633 val accuracy: 0.426000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.401286 val accuracy: 0.396000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.396816 val accuracy: 0.377000
best validation accuracy achieved during cross-validation: 0.426000
```

[5]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.418
```

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

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



1.3.1 Inline question 1:

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

Your Answer :

TODO answer the question

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)
```

[108]:
```
from cs231n.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 128
num_classes = 10


learning_rates = [0.3]
regularization_strengths = [0.001]
results = {}
################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
    for reg in regularization_strengths:
        net = TwoLayerNet(input_dim, hidden_dim, num_classes)
        net.train(X = X_train_feats, y = y_train,
                  num_iters=2000, batch_size=200,
                  learning_rate=lr, learning_rate_decay=0.85,
                  reg=reg, verbose=True)

        y_train_pred = net.predict(X_train_feats)
        train_accuracy = np.mean(y_train == y_train_pred)
        results[(lr, reg)] = train_accuracy
pass

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Print out results.
for lr, reg in sorted(results):
    train_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f' % (lr, reg, train_accuracy))

best_net = net
```

```
iteration 0 / 2000: loss 2.302602
iteration 100 / 2000: loss 1.745171
iteration 200 / 2000: loss 1.277187
iteration 300 / 2000: loss 1.405192
iteration 400 / 2000: loss 1.315692
iteration 500 / 2000: loss 1.378686
iteration 600 / 2000: loss 1.329843
iteration 700 / 2000: loss 1.396741
iteration 800 / 2000: loss 1.450456
iteration 900 / 2000: loss 1.254494
iteration 1000 / 2000: loss 1.305599
iteration 1100 / 2000: loss 1.260513
iteration 1200 / 2000: loss 1.154122
iteration 1300 / 2000: loss 1.332968
iteration 1400 / 2000: loss 1.182166
iteration 1500 / 2000: loss 1.196408
iteration 1600 / 2000: loss 1.209150
iteration 1700 / 2000: loss 1.087451
iteration 1800 / 2000: loss 1.302890
iteration 1900 / 2000: loss 1.252634
lr 3.000000e-01 reg 1.000000e-03 train accuracy: 0.606122
```

1.4.1 Note added by the writer:

It should use Cross-Validation instructed by the homework.

[109]:
```
print(np.sqrt(np.sum(net.params['W1']**2))+
      np.sqrt(np.sum(net.params['W2']**2))+
      np.sqrt(np.sum(net.params['b1']**2))+
      np.sqrt(np.sum(net.params['b2']**2)))
```

```
14.277640834444636
```

[110]:
```
# 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)
```

```
0.567
```

[111]:
```
-np.log(0.1)
```

[111]:
```
2.3025850929940455
```

2 Note added by the writer

As the newcomer to the CNN, at first I train the network using the small learning_rate(<0.0001), actually it seems well in reinforcement learning. I got the near constant loss and was confused for serval hours. The loss is 2.302585, nearly equal to $-log_{10}0.1$. But I didn't realize at the beginning. So it is a lesson that always check the weight and loss in the network. mark!!