In [1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment1/'
# FOLDERNAME = 'assignment1'
# assert FOLDERNAME is not None, "[!] Enter the foldername."

# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME

# # Install requirements from colab_requirements.txt
# # TODO: Please change your path below to the colab_requirements.txt file
# ! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/colab_requirements
```

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 transferring 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.

In [2]:
```
# Run some setup code for this notebook.
from __future__ import print_function

import random
import numpy as np
from cs6353.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
```

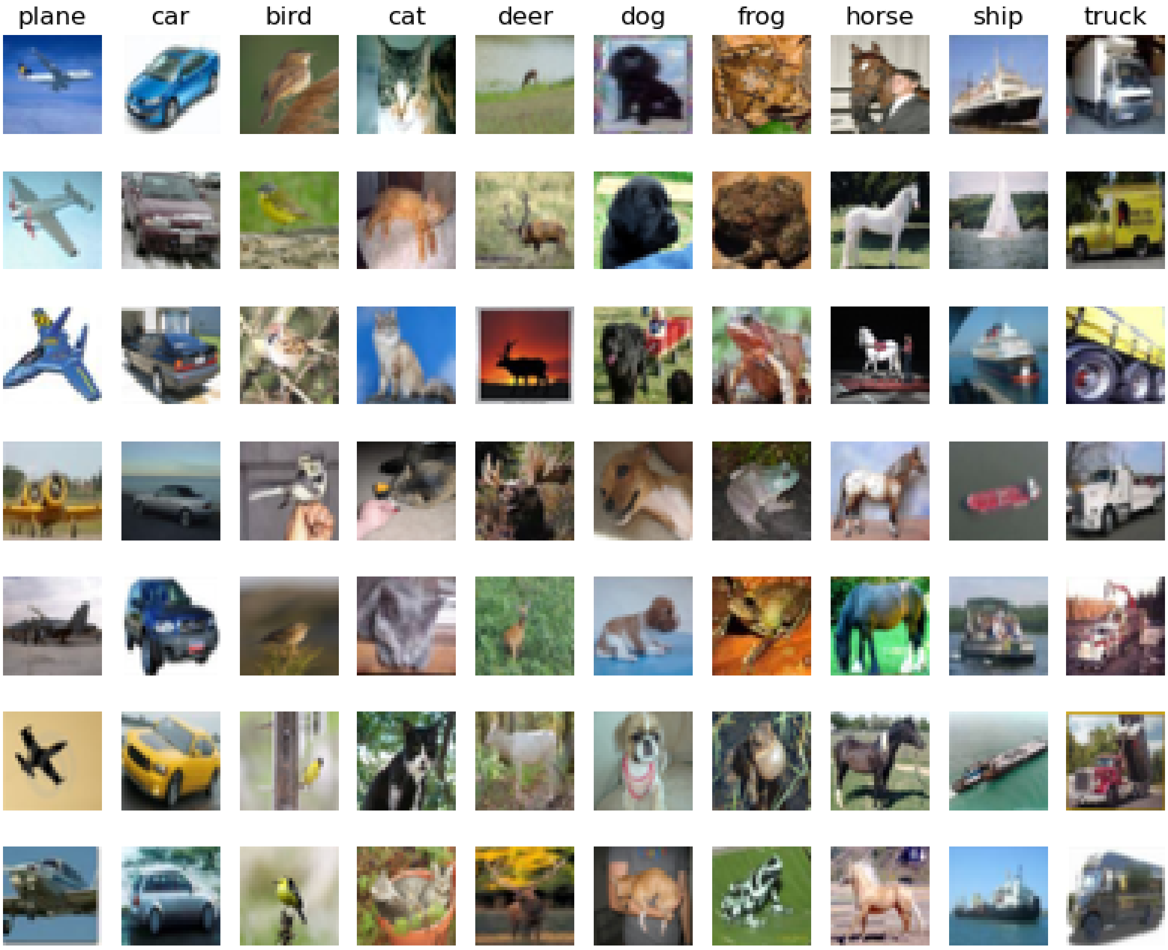
In [3]:

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs6353/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause memo
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,)
```

In [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', 't
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()
```



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

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

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

In [7]:
```
from cs6353.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 `cs6353/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.

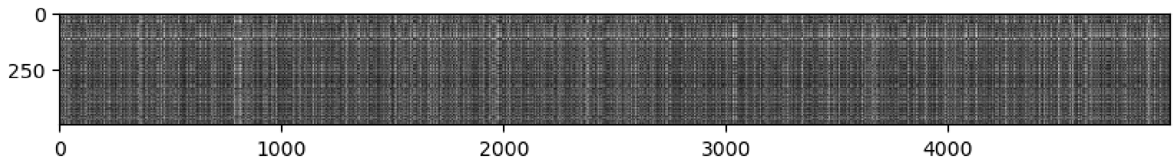
In [8]:

```
# Open cs6353/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)
```

In [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?

Your Answer:

1. The reason for the bright rows is that the test image is very different from most or all of the training images.
2. The reason for the bright columns is that the training image is very different from most or all of the images used for testing.

In [10]:

```
# 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`:

In [11]:
```
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. (Mean and standard deviation in (1) and (2) are vectors and can be different across dimensions)

Your Answer: 1 and 2

Your explanation: L1 distance = d(I1,I2) = sigma_p|I1^p-I2^p|

1. For L1 distance, subtracting the mean can be represented as sigma_p|(I1^p - u) - (I2^p - u)| which u is the mean of the image. However, this tends to represent the same order as the order that not do anything because this is = sigma_p|I1^p - I2^p|;
2. For L1 distance, subtracting the mean can be represented as sigma_p|(I1^p - u) / sqrt(Var(I)) - (I2^p - u) / sqrt(Var(I))| which u is the mean of the image and sqrt(Var(I)) is the standard deviation. However, this tends to represent the same order as the order that not do anything because this is = (sigma_p|I1^p - I2^p|) / sqrt(Var(I));
3. It will change the order after rotation a degree such as 30 or 45

In [12]:
```
# 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
```

In [13]:
```
# 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
```

In [14]:
```
# 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 exe
    """
    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 implemen
```

```
Two loop version took 23.926166 seconds
One loop version took 39.427992 seconds
No loop version took 0.145675 seconds
```

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.

In [15]:
```
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)
```

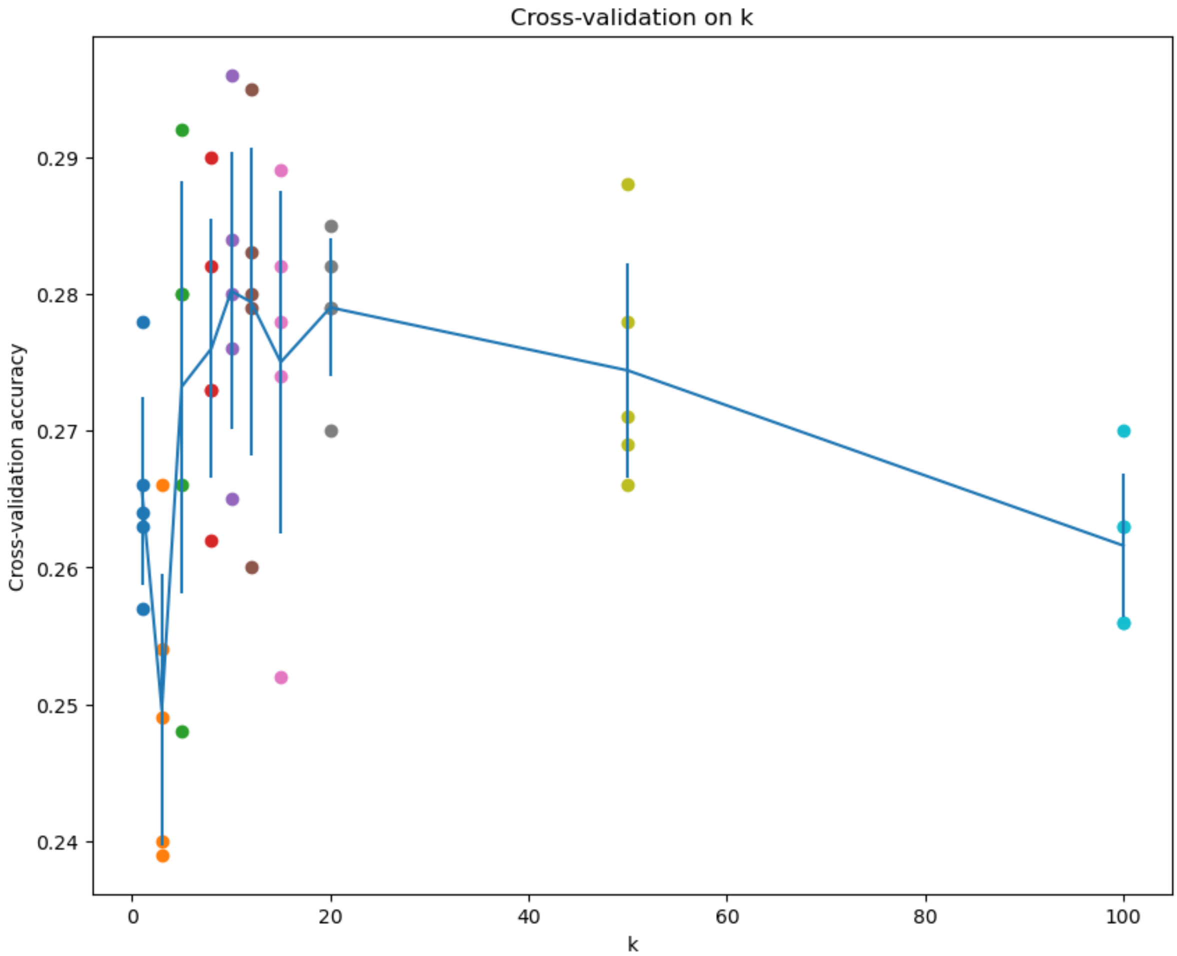
```
##############################################################################
#                                 END OF YOUR CODE                           #
##############################################################################

# 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 number in k_choices:
    k_to_accuracies[number] = []
    for i in range(num_folds):
        new_X_train_folds = []
        new_y_train_folds = []
        for index in range(num_folds):
            if index != i:
                new_X_train_folds.append(X_train_folds[index])
                new_y_train_folds.append(y_train_folds[index])
        new_X_train = np.vstack(new_X_train_folds)
        new_y_train = np.concatenate(new_y_train_folds)
        X_validation = X_train_folds[i]
        y_validation = y_train_folds[i]
        classifier = KNearestNeighbor()
        classifier.train(new_X_train, new_y_train)
        validation_dists = classifier.compute_distances_no_loops(X_validation)
        y_validation_pred = classifier.predict_labels(validation_dists, number)
        num_correct = np.sum(y_validation_pred == y_validation)
        accuracy = float(num_correct) / len(y_validation)
        k_to_accuracies[number].append(accuracy)

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

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

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

In [16]:

```
# 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()
```



In [17]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3 Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply.

1. The training error of a 1-NN will always be better than or equal to 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 and 4

Your explanation:

1. Because when tested with the training set, 1-NN will always have an image with distance 0 (identical to it), and the error is 0. But when using 5-NN, this exact same image only accounts for all 1 /5, which results in the possibility of ending up with a different class. Therefore for 5-NN, 0 error is the upper limit.
2. Wrong, the practical example just showed that 5-NN may be better than 1-NN for the test set.
3. K-NN is not linear classification
4. This is correct, because K-NN needs to calculate the distance between the test set and each picture in the training set when predicting, so when the training set becomes larger, the amount of calculation increases and the time required for classification increases.

In [1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment1/'
# FOLDERNAME = 'assignment1'
# assert FOLDERNAME is not None, "[!] Enter the foldername."

# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME

# # Install requirements from colab_requirements.txt
# # TODO: Please change your path below to the colab_requirements.txt file
# ! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/colab_requirements
```

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

In [2]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs6353.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
```

CIFAR-10 Data Loading and Preprocessing

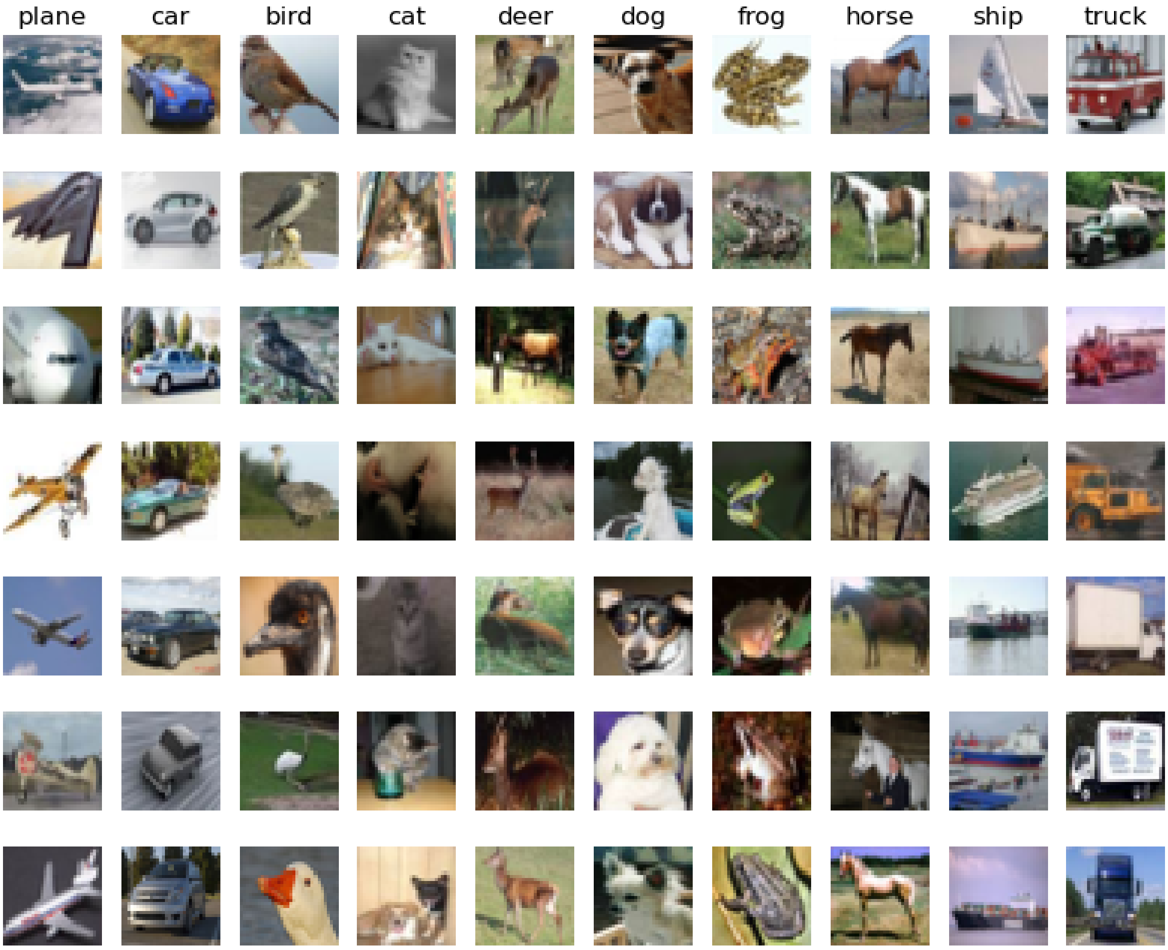
In [3]:

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs6353/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause memo
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,)
```

In [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', 't
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()
```



In [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('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,)
```

In [6]:

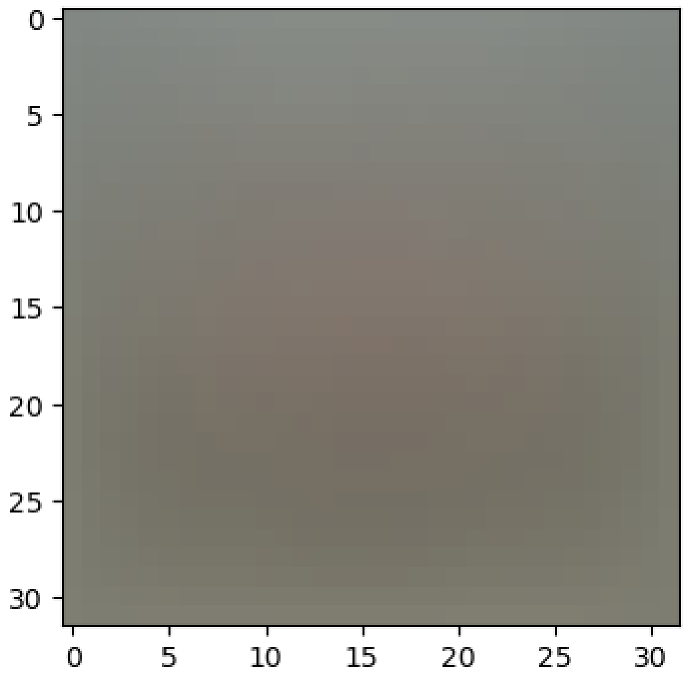
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

In [7]:

```
# 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]
```



In [8]:

```
# 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
```

In [9]:

```
# 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)
```

SVM Classifier

Your code for this section will all be written inside **cs6353/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.

In [10]:

```
# Evaluate the naive implementation of the loss we provided for you:
from cs6353.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: 8.811875
```

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:

In [11]:

```
# 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 cs6353.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: 25.804470 analytic: 25.804470, relative error: 9.852116e-12
numerical: -7.431729 analytic: -7.431729, relative error: 2.767831e-11
numerical: -42.778346 analytic: -42.778346, relative error: 8.387403e-12
numerical: -56.789906 analytic: -56.789906, relative error: 5.495111e-12
numerical: -21.750375 analytic: -21.750375, relative error: 4.170569e-12
numerical: 0.127673 analytic: 0.127673, relative error: 5.459872e-10
numerical: -7.641180 analytic: -7.641180, relative error: 6.354174e-11
numerical: 0.144840 analytic: 0.144840, relative error: 7.539416e-10
numerical: -14.386256 analytic: -14.386256, relative error: 1.095473e-11
numerical: 2.962742 analytic: 2.962742, relative error: 1.389373e-11
numerical: -5.243442 analytic: -5.243442, relative error: 7.553114e-12
numerical: -30.148099 analytic: -30.148099, relative error: 1.108405e-11
numerical: 23.482853 analytic: 23.482853, relative error: 1.564578e-11
numerical: -1.747361 analytic: -1.747361, relative error: 4.969919e-11
numerical: -9.517579 analytic: -9.517579, relative error: 6.387553e-12
numerical: 2.785900 analytic: 2.785900, relative error: 3.140856e-12
numerical: 10.325072 analytic: 10.325072, relative error: 1.038202e-11
numerical: 24.532150 analytic: 24.532150, relative error: 7.337818e-12
numerical: 17.487602 analytic: 17.487602, relative error: 2.679934e-12
numerical: -24.462290 analytic: -24.462290, relative error: 7.996779e-12
```

Inline Question 1:

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

Your Answer:

1. Because the loss function of SVM uses max, it is not a continuous function, so it is strictly not differentiable.
2. This is not a cause for concern, since the gradient at points other than discontinuities can be found unaffected.
3. When Si - Syi + delta = 0, it fails.
4. Increasing delta can improve this situation.

In [12]:
```
# 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 cs6353.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.811875e+00 computed in 0.037381s
Vectorized loss: 8.811875e+00 computed in 0.001994s
difference: 0.000000
```

In [13]:

```
# 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 0.035880s
Vectorized loss and gradient: computed in 0.001995s
difference: 0.000000
```

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.

In [14]:

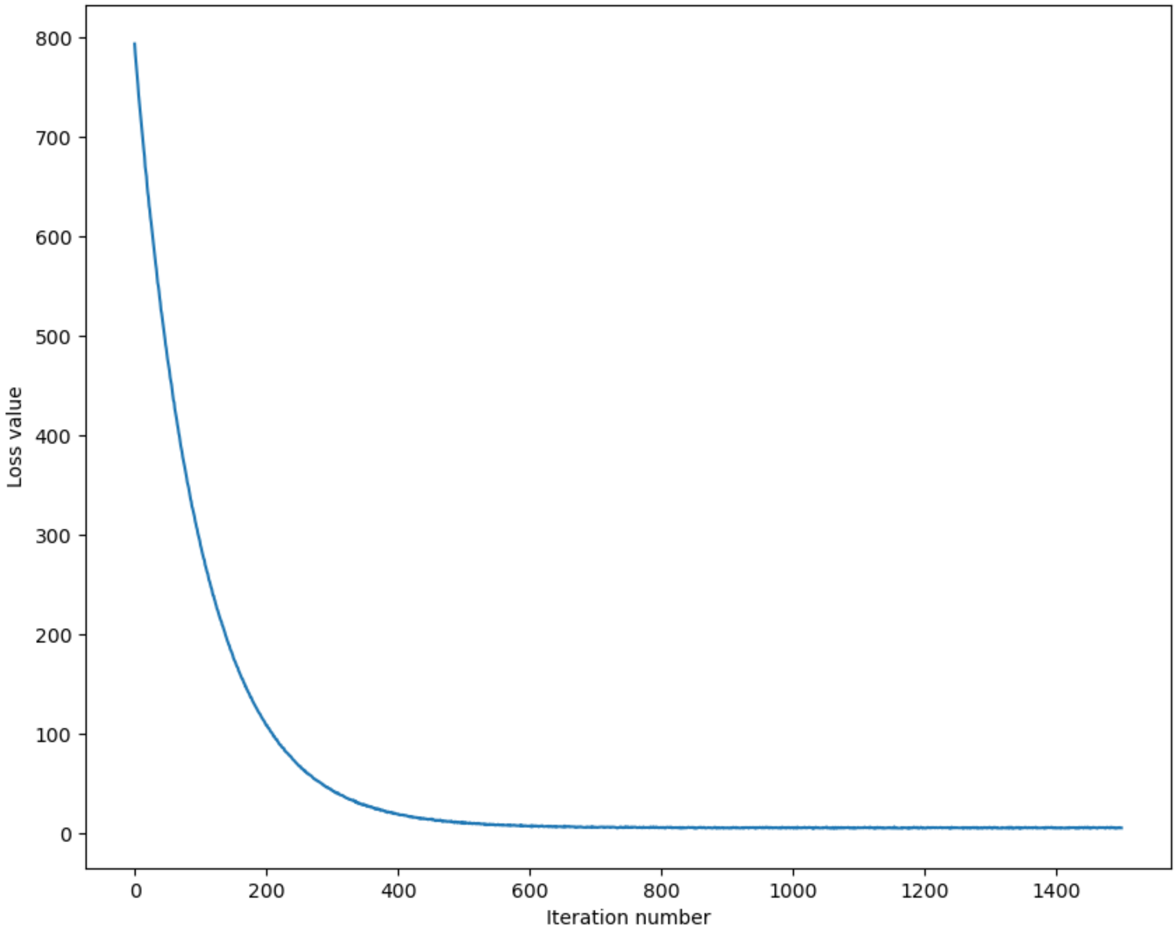
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs6353.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 793.992839
iteration 100 / 1500: loss 289.524506
iteration 200 / 1500: loss 108.542615
iteration 300 / 1500: loss 42.855749
iteration 400 / 1500: loss 18.933640
iteration 500 / 1500: loss 9.884677
iteration 600 / 1500: loss 7.078068
iteration 700 / 1500: loss 5.795904
iteration 800 / 1500: loss 5.426649
iteration 900 / 1500: loss 5.157825
iteration 1000 / 1500: loss 5.644373
iteration 1100 / 1500: loss 5.328745
iteration 1200 / 1500: loss 5.207528
iteration 1300 / 1500: loss 5.278697
iteration 1400 / 1500: loss 5.160449
That took 3.406090s
```

In [15]:

```
# 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()
```



In [16]:
```
# 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: 0.373061
validation accuracy: 0.390000
```

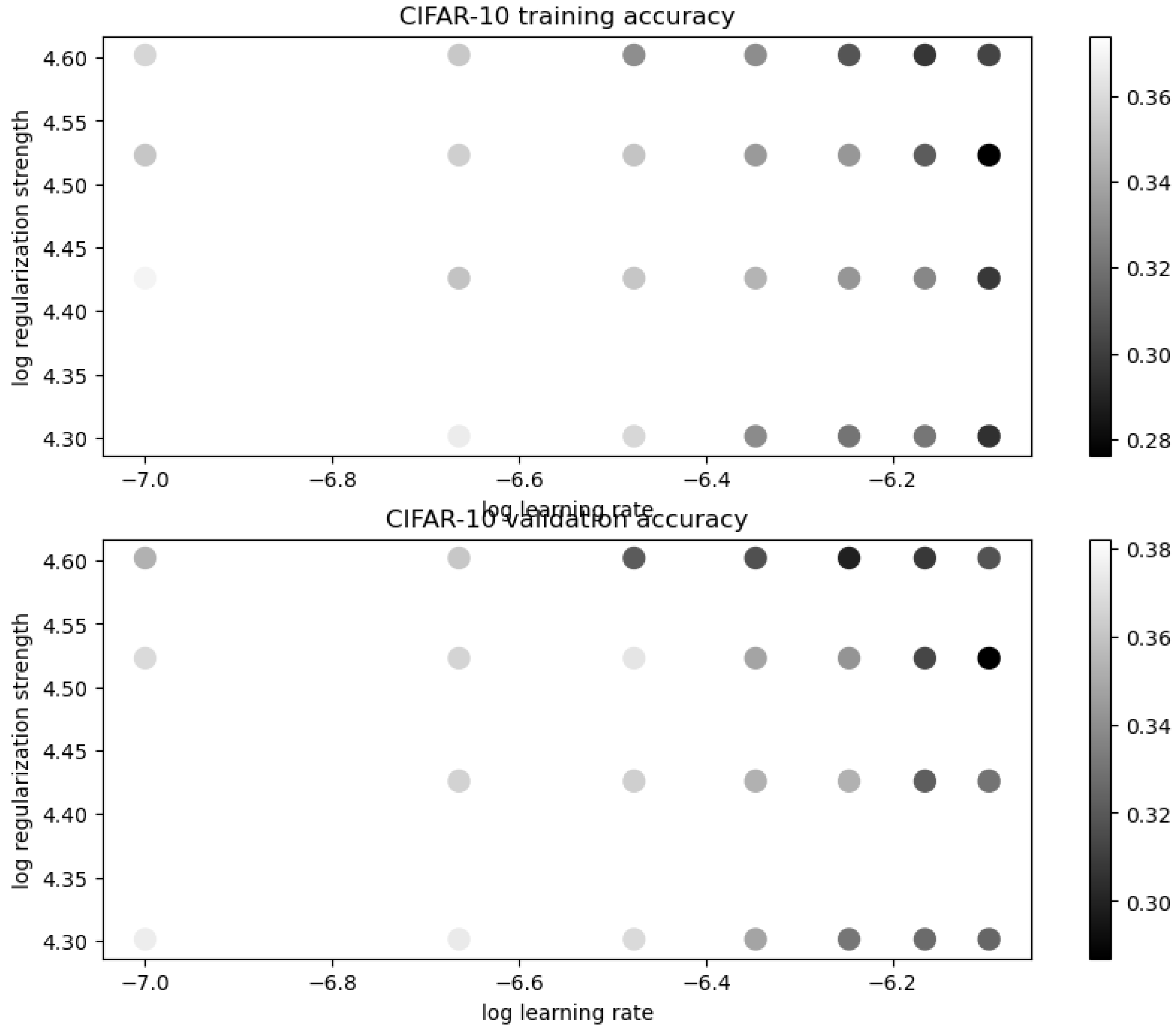
In [17]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = np.linspace(1e-7, 8e-7, 7)
regularization_strengths = np.linspace(2e4, 4e4, 4)

# 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.                                      #
################################################################################

for learn in learning_rates:
    for regular in regularization_strengths:
        new_svm = LinearSVM()
        new_svm.train(X_train, y_train, learn, regular, num_iters=1500, verbose=Fal
        y_train_pred = new_svm.predict(X_train)
        y_val_pred = new_svm.predict(X_val)
        train_acc = np.mean(y_train == y_train_pred)
        val_acc = np.mean(y_val == y_val_pred)
        if best_val < val_acc:
            best_val = val_acc
            best_svm = new_svm
        results[(learn, regular)] = (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)
```

```
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.374061 val accuracy: 0.375000
lr 1.000000e-07 reg 2.666667e+04 train accuracy: 0.369571 val accuracy: 0.382000
lr 1.000000e-07 reg 3.333333e+04 train accuracy: 0.352082 val accuracy: 0.368000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.358204 val accuracy: 0.353000
lr 2.166667e-07 reg 2.000000e+04 train accuracy: 0.366449 val accuracy: 0.374000
lr 2.166667e-07 reg 2.666667e+04 train accuracy: 0.350735 val accuracy: 0.365000
lr 2.166667e-07 reg 3.333333e+04 train accuracy: 0.355510 val accuracy: 0.366000
lr 2.166667e-07 reg 4.000000e+04 train accuracy: 0.352653 val accuracy: 0.361000
lr 3.333333e-07 reg 2.000000e+04 train accuracy: 0.358673 val accuracy: 0.368000
lr 3.333333e-07 reg 2.666667e+04 train accuracy: 0.352061 val accuracy: 0.364000
lr 3.333333e-07 reg 3.333333e+04 train accuracy: 0.351327 val accuracy: 0.372000
lr 3.333333e-07 reg 4.000000e+04 train accuracy: 0.330469 val accuracy: 0.321000
lr 4.500000e-07 reg 2.000000e+04 train accuracy: 0.329551 val accuracy: 0.348000
lr 4.500000e-07 reg 2.666667e+04 train accuracy: 0.345082 val accuracy: 0.353000
lr 4.500000e-07 reg 3.333333e+04 train accuracy: 0.335102 val accuracy: 0.348000
lr 4.500000e-07 reg 4.000000e+04 train accuracy: 0.330408 val accuracy: 0.317000
lr 5.666667e-07 reg 2.000000e+04 train accuracy: 0.320878 val accuracy: 0.331000
lr 5.666667e-07 reg 2.666667e+04 train accuracy: 0.333857 val accuracy: 0.353000
lr 5.666667e-07 reg 3.333333e+04 train accuracy: 0.334224 val accuracy: 0.342000
lr 5.666667e-07 reg 4.000000e+04 train accuracy: 0.308755 val accuracy: 0.299000
lr 6.833333e-07 reg 2.000000e+04 train accuracy: 0.321592 val accuracy: 0.327000
lr 6.833333e-07 reg 2.666667e+04 train accuracy: 0.328041 val accuracy: 0.322000
lr 6.833333e-07 reg 3.333333e+04 train accuracy: 0.311612 val accuracy: 0.313000
lr 6.833333e-07 reg 4.000000e+04 train accuracy: 0.297163 val accuracy: 0.308000
lr 8.000000e-07 reg 2.000000e+04 train accuracy: 0.294633 val accuracy: 0.325000
lr 8.000000e-07 reg 2.666667e+04 train accuracy: 0.297857 val accuracy: 0.330000
lr 8.000000e-07 reg 3.333333e+04 train accuracy: 0.276143 val accuracy: 0.287000
lr 8.000000e-07 reg 4.000000e+04 train accuracy: 0.302531 val accuracy: 0.318000
best validation accuracy achieved during cross-validation: 0.382000
```

In [18]:

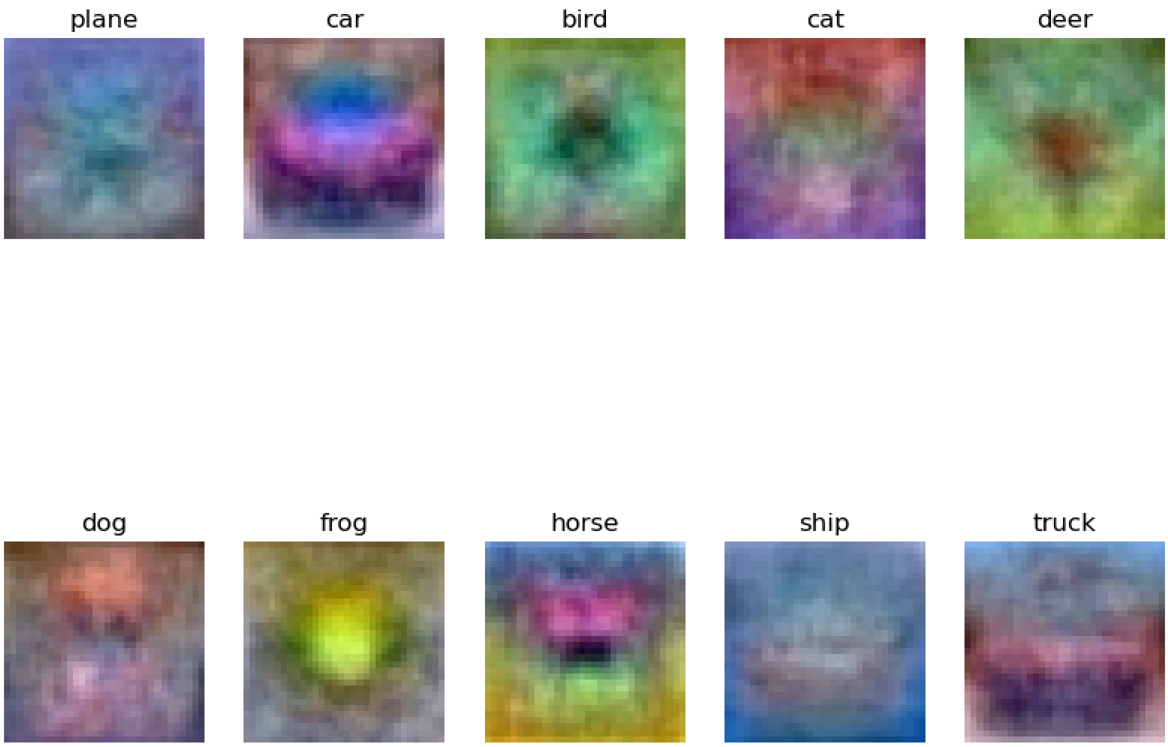
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 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()
```



In [19]:
```
# 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: 0.367000
```

In [20]:
```
# 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', 't
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



Inline question 2:

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

Your answer:

1. Visualized weights have similar primary colors and outlines to class objects.
2. SVM continuously updates the template through gradient descent to make the L2 distance between the template and the training image shorter, making the template similar to each training image of the category.

```
In [ ]:
```

In [1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment1/'
# FOLDERNAME = 'assignment1'
# assert FOLDERNAME is not None, "[!] Enter the foldername."

# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME

# # Install requirements from colab_requirements.txt
# # TODO: Please change your path below to the colab_requirements.txt file
# ! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/colab_requirements
```

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

In [2]:
```
from __future__ import print_function
import random
import numpy as np
from cs6353.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 external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
```

```
%load_ext autoreload
%autoreload 2
```

In [3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_de
    """
    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 = 'cs6353/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 memo
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,)
```

Softmax Classifier

Your code for this section will all be written inside **cs6353/classifiers/softmax.py**.

In [4]:

```
# First implement the naive softmax loss function with nested loops.
# Open the file cs6353/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs6353.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: 2.398060
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: Because for the weights without gradient descent, it can be roughly estimated that the probability of selecting each category is 1/10, so the probability of selecting the correct category is 1/10. The loss function can be expressed as 1/N * sigma (-log (1 /10)), its value is -log(0.1).

In [5]:

```
# 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 cs6353.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
```

```
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.672178 analytic: -0.672178, relative error: 4.682864e-08
numerical: 6.184807 analytic: 6.184807, relative error: 1.222230e-08
numerical: 0.186668 analytic: 0.186668, relative error: 1.368333e-08
numerical: -1.766186 analytic: -1.766186, relative error: 3.407252e-08
numerical: -3.437839 analytic: -3.437839, relative error: 2.103722e-08
numerical: -2.310615 analytic: -2.310615, relative error: 7.520068e-09
numerical: -0.041020 analytic: -0.041020, relative error: 2.381192e-07
numerical: -2.044117 analytic: -2.044117, relative error: 1.198714e-08
numerical: -0.944077 analytic: -0.944077, relative error: 8.487483e-09
numerical: -0.915686 analytic: -0.915686, relative error: 6.271561e-08
numerical: 0.622275 analytic: 0.622275, relative error: 4.265229e-08
numerical: -2.839514 analytic: -2.839514, relative error: 4.901588e-09
numerical: 0.842350 analytic: 0.842350, relative error: 7.881849e-08
numerical: 2.091051 analytic: 2.091051, relative error: 3.783442e-08
numerical: 1.650637 analytic: 1.650637, relative error: 2.544628e-09
numerical: 0.583131 analytic: 0.583131, relative error: 1.168748e-08
numerical: 0.776396 analytic: 0.776396, relative error: 5.608973e-08
numerical: 1.974616 analytic: 1.974616, relative error: 5.357516e-09
numerical: 2.304291 analytic: 2.304291, relative error: 2.288593e-08
numerical: 0.606504 analytic: 0.606504, relative error: 9.107837e-08
```

In [6]:

```
# Now that we have a naive implementation of the softmax loss function and its gradi
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version shoul
# 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 cs6353.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.398060e+00 computed in 0.049833s
vectorized loss: 2.398060e+00 computed in 0.001995s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [7]:

```
# 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 cs6353.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = np.linspace(8e-8, 1e-6, 10)
regularization_strengths = np.linspace(1.5e4, 4e4, 10)
# 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 classifier in best_softmax.                          #
################################################################################

for learn in learning_rates:
    for regular in regularization_strengths:
        new_softmax = Softmax()
        new_softmax.train(X_train, y_train, learn, regular, num_iters=1500, verbose=
        y_train_pred = new_softmax.predict(X_train)
        y_val_pred = new_softmax.predict(X_val)
        train_acc = np.mean(y_train == y_train_pred)
        val_acc = np.mean(y_val == y_val_pred)
        if best_val < val_acc:
            best_val = val_acc
            best_softmax = new_softmax
        results[(learn, regular)] = (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)
```

```
iteration 0 / 1500: loss 464.768182
iteration 100 / 1500: loss 286.698064
iteration 200 / 1500: loss 177.403418
iteration 300 / 1500: loss 110.294864
iteration 400 / 1500: loss 69.049094
iteration 500 / 1500: loss 43.236798
iteration 600 / 1500: loss 27.565002
iteration 700 / 1500: loss 17.752926
iteration 800 / 1500: loss 11.722581
iteration 900 / 1500: loss 8.017363
iteration 1000 / 1500: loss 5.702425
iteration 1100 / 1500: loss 4.319076
iteration 1200 / 1500: loss 3.529038
iteration 1300 / 1500: loss 2.914610
iteration 1400 / 1500: loss 2.609000
iteration 0 / 1500: loss 554.824261
iteration 100 / 1500: loss 313.484443
iteration 200 / 1500: loss 177.729218
iteration 300 / 1500: loss 101.170909
iteration 400 / 1500: loss 58.176490
iteration 500 / 1500: loss 33.683133
iteration 600 / 1500: loss 19.899195
iteration 700 / 1500: loss 12.156976
iteration 800 / 1500: loss 7.766400
iteration 900 / 1500: loss 5.292095
iteration 1000 / 1500: loss 3.828682
iteration 1100 / 1500: loss 3.187020
iteration 1200 / 1500: loss 2.577394
iteration 1300 / 1500: loss 2.375488
iteration 1400 / 1500: loss 2.253092
iteration 0 / 1500: loss 633.846836
iteration 100 / 1500: loss 327.290989
iteration 200 / 1500: loss 170.149965
iteration 300 / 1500: loss 88.752268
iteration 400 / 1500: loss 46.773606
iteration 500 / 1500: loss 25.226816
iteration 600 / 1500: loss 14.025741
iteration 700 / 1500: loss 8.287788
iteration 800 / 1500: loss 5.290053
iteration 900 / 1500: loss 3.701510
iteration 1000 / 1500: loss 2.959902
iteration 1100 / 1500: loss 2.518391
iteration 1200 / 1500: loss 2.309659
iteration 1300 / 1500: loss 2.171430
iteration 1400 / 1500: loss 2.158646
iteration 0 / 1500: loss 716.833864
iteration 100 / 1500: loss 339.064600
iteration 200 / 1500: loss 161.051337
iteration 300 / 1500: loss 77.205590
iteration 400 / 1500: loss 37.512041
iteration 500 / 1500: loss 18.892142
iteration 600 / 1500: loss 9.997532
iteration 700 / 1500: loss 5.812813
iteration 800 / 1500: loss 3.873121
iteration 900 / 1500: loss 2.935442
iteration 1000 / 1500: loss 2.473535
iteration 1100 / 1500: loss 2.248061
iteration 1200 / 1500: loss 2.143920
iteration 1300 / 1500: loss 2.150002
iteration 1400 / 1500: loss 2.048683
iteration 0 / 1500: loss 812.891761
iteration 100 / 1500: loss 351.896349
iteration 200 / 1500: loss 152.982942
iteration 300 / 1500: loss 67.338754
```

```
iteration 400 / 1500: loss 30.262149
iteration 500 / 1500: loss 14.251660
iteration 600 / 1500: loss 7.344385
iteration 700 / 1500: loss 4.364301
iteration 800 / 1500: loss 3.039987
iteration 900 / 1500: loss 2.551947
iteration 1000 / 1500: loss 2.238912
iteration 1100 / 1500: loss 2.088885
iteration 1200 / 1500: loss 2.081459
iteration 1300 / 1500: loss 2.117538
iteration 1400 / 1500: loss 2.084181
iteration 0 / 1500: loss 887.517976
iteration 100 / 1500: loss 351.721960
iteration 200 / 1500: loss 140.162291
iteration 300 / 1500: loss 56.653437
iteration 400 / 1500: loss 23.624119
iteration 500 / 1500: loss 10.558423
iteration 600 / 1500: loss 5.449196
iteration 700 / 1500: loss 3.435088
iteration 800 / 1500: loss 2.549853
iteration 900 / 1500: loss 2.299047
iteration 1000 / 1500: loss 2.173079
iteration 1100 / 1500: loss 2.174168
iteration 1200 / 1500: loss 2.119393
iteration 1300 / 1500: loss 2.139666
iteration 1400 / 1500: loss 2.098097
iteration 0 / 1500: loss 982.139109
iteration 100 / 1500: loss 355.772479
iteration 200 / 1500: loss 129.818740
iteration 300 / 1500: loss 48.225440
iteration 400 / 1500: loss 18.811415
iteration 500 / 1500: loss 8.113041
iteration 600 / 1500: loss 4.273060
iteration 700 / 1500: loss 2.842279
iteration 800 / 1500: loss 2.437034
iteration 900 / 1500: loss 2.225387
iteration 1000 / 1500: loss 2.114433
iteration 1100 / 1500: loss 2.107111
iteration 1200 / 1500: loss 2.114422
iteration 1300 / 1500: loss 2.112970
iteration 1400 / 1500: loss 2.106965
iteration 0 / 1500: loss 1060.172995
iteration 100 / 1500: loss 350.991114
iteration 200 / 1500: loss 117.388988
iteration 300 / 1500: loss 40.212523
iteration 400 / 1500: loss 14.702280
iteration 500 / 1500: loss 6.311611
iteration 600 / 1500: loss 3.532658
iteration 700 / 1500: loss 2.574163
iteration 800 / 1500: loss 2.231367
iteration 900 / 1500: loss 2.194532
iteration 1000 / 1500: loss 2.160525
iteration 1100 / 1500: loss 2.096662
iteration 1200 / 1500: loss 2.132063
iteration 1300 / 1500: loss 2.138003
iteration 1400 / 1500: loss 2.133870
iteration 0 / 1500: loss 1146.002206
iteration 100 / 1500: loss 347.233953
iteration 200 / 1500: loss 106.351839
iteration 300 / 1500: loss 33.631259
iteration 400 / 1500: loss 11.614714
iteration 500 / 1500: loss 4.944960
iteration 600 / 1500: loss 2.942938
iteration 700 / 1500: loss 2.394537
```

```
iteration 800 / 1500: loss 2.184364
iteration 900 / 1500: loss 2.112868
iteration 1000 / 1500: loss 2.095780
iteration 1100 / 1500: loss 2.143291
iteration 1200 / 1500: loss 2.119253
iteration 1300 / 1500: loss 2.077600
iteration 1400 / 1500: loss 2.147264
iteration 0 / 1500: loss 1243.643808
iteration 100 / 1500: loss 344.732867
iteration 200 / 1500: loss 96.699653
iteration 300 / 1500: loss 28.234977
iteration 400 / 1500: loss 9.293539
iteration 500 / 1500: loss 4.094677
iteration 600 / 1500: loss 2.674493
iteration 700 / 1500: loss 2.273210
iteration 800 / 1500: loss 2.190165
iteration 900 / 1500: loss 2.139674
iteration 1000 / 1500: loss 2.172040
iteration 1100 / 1500: loss 2.141595
iteration 1200 / 1500: loss 2.141884
iteration 1300 / 1500: loss 2.151340
iteration 1400 / 1500: loss 2.137915
iteration 0 / 1500: loss 468.695781
iteration 100 / 1500: loss 157.073942
iteration 200 / 1500: loss 53.651679
iteration 300 / 1500: loss 19.173686
iteration 400 / 1500: loss 7.700028
iteration 500 / 1500: loss 3.912463
iteration 600 / 1500: loss 2.569832
iteration 700 / 1500: loss 2.249175
iteration 800 / 1500: loss 2.075259
iteration 900 / 1500: loss 2.044183
iteration 1000 / 1500: loss 2.153326
iteration 1100 / 1500: loss 1.986222
iteration 1200 / 1500: loss 2.050826
iteration 1300 / 1500: loss 2.017789
iteration 1400 / 1500: loss 2.050729
iteration 0 / 1500: loss 549.270756
iteration 100 / 1500: loss 150.178634
iteration 200 / 1500: loss 42.174028
iteration 300 / 1500: loss 12.943574
iteration 400 / 1500: loss 4.977956
iteration 500 / 1500: loss 2.831300
iteration 600 / 1500: loss 2.381308
iteration 700 / 1500: loss 2.160887
iteration 800 / 1500: loss 2.054446
iteration 900 / 1500: loss 2.032254
iteration 1000 / 1500: loss 1.999281
iteration 1100 / 1500: loss 2.010083
iteration 1200 / 1500: loss 2.054578
iteration 1300 / 1500: loss 2.023151
iteration 1400 / 1500: loss 2.120635
iteration 0 / 1500: loss 632.464719
iteration 100 / 1500: loss 141.020580
iteration 200 / 1500: loss 32.779895
iteration 300 / 1500: loss 8.870639
iteration 400 / 1500: loss 3.658188
iteration 500 / 1500: loss 2.402646
iteration 600 / 1500: loss 2.156470
iteration 700 / 1500: loss 2.038050
iteration 800 / 1500: loss 2.045151
iteration 900 / 1500: loss 2.101272
iteration 1000 / 1500: loss 2.077290
iteration 1100 / 1500: loss 2.157673
```

```
iteration 1200 / 1500: loss 2.035729
iteration 1300 / 1500: loss 2.079979
iteration 1400 / 1500: loss 2.110386
iteration 0 / 1500: loss 725.939278
iteration 100 / 1500: loss 132.486968
iteration 200 / 1500: loss 25.642880
iteration 300 / 1500: loss 6.326133
iteration 400 / 1500: loss 2.890985
iteration 500 / 1500: loss 2.219470
iteration 600 / 1500: loss 2.039387
iteration 700 / 1500: loss 2.051384
iteration 800 / 1500: loss 2.029789
iteration 900 / 1500: loss 2.150457
iteration 1000 / 1500: loss 2.119879
iteration 1100 / 1500: loss 2.103512
iteration 1200 / 1500: loss 2.102287
iteration 1300 / 1500: loss 2.095328
iteration 1400 / 1500: loss 2.050472
iteration 0 / 1500: loss 804.980970
iteration 100 / 1500: loss 119.870146
iteration 200 / 1500: loss 19.338106
iteration 300 / 1500: loss 4.641577
iteration 400 / 1500: loss 2.451336
iteration 500 / 1500: loss 2.194118
iteration 600 / 1500: loss 2.102567
iteration 700 / 1500: loss 2.093923
iteration 800 / 1500: loss 2.093202
iteration 900 / 1500: loss 2.107619
iteration 1000 / 1500: loss 2.048011
iteration 1100 / 1500: loss 2.115489
iteration 1200 / 1500: loss 2.090407
iteration 1300 / 1500: loss 2.121482
iteration 1400 / 1500: loss 2.043491
iteration 0 / 1500: loss 890.732222
iteration 100 / 1500: loss 108.291266
iteration 200 / 1500: loss 14.814257
iteration 300 / 1500: loss 3.576812
iteration 400 / 1500: loss 2.296516
iteration 500 / 1500: loss 2.096252
iteration 600 / 1500: loss 2.106150
iteration 700 / 1500: loss 2.117343
iteration 800 / 1500: loss 2.092747
iteration 900 / 1500: loss 2.067513
iteration 1000 / 1500: loss 2.100020
iteration 1100 / 1500: loss 2.078335
iteration 1200 / 1500: loss 2.062025
iteration 1300 / 1500: loss 2.084377
iteration 1400 / 1500: loss 2.131117
iteration 0 / 1500: loss 986.392180
iteration 100 / 1500: loss 98.027497
iteration 200 / 1500: loss 11.487112
iteration 300 / 1500: loss 3.033591
iteration 400 / 1500: loss 2.184153
iteration 500 / 1500: loss 2.070350
iteration 600 / 1500: loss 2.139172
iteration 700 / 1500: loss 2.062363
iteration 800 / 1500: loss 2.071719
iteration 900 / 1500: loss 2.189918
iteration 1000 / 1500: loss 2.130251
iteration 1100 / 1500: loss 2.136630
iteration 1200 / 1500: loss 2.054111
iteration 1300 / 1500: loss 2.116912
iteration 1400 / 1500: loss 2.084189
iteration 0 / 1500: loss 1072.229887
```

```
iteration 100 / 1500: loss 86.986798
iteration 200 / 1500: loss 8.913776
iteration 300 / 1500: loss 2.652950
iteration 400 / 1500: loss 2.116437
iteration 500 / 1500: loss 2.084276
iteration 600 / 1500: loss 2.130919
iteration 700 / 1500: loss 2.113057
iteration 800 / 1500: loss 2.113139
iteration 900 / 1500: loss 2.120281
iteration 1000 / 1500: loss 2.167142
iteration 1100 / 1500: loss 2.085689
iteration 1200 / 1500: loss 2.143927
iteration 1300 / 1500: loss 2.101340
iteration 1400 / 1500: loss 2.087509
iteration 0 / 1500: loss 1134.448560
iteration 100 / 1500: loss 75.337460
iteration 200 / 1500: loss 6.922764
iteration 300 / 1500: loss 2.425643
iteration 400 / 1500: loss 2.174435
iteration 500 / 1500: loss 2.095884
iteration 600 / 1500: loss 2.119742
iteration 700 / 1500: loss 2.088575
iteration 800 / 1500: loss 2.106962
iteration 900 / 1500: loss 2.158853
iteration 1000 / 1500: loss 2.121461
iteration 1100 / 1500: loss 2.124882
iteration 1200 / 1500: loss 2.117246
iteration 1300 / 1500: loss 2.110526
iteration 1400 / 1500: loss 2.109461
iteration 0 / 1500: loss 1239.435096
iteration 100 / 1500: loss 67.281834
iteration 200 / 1500: loss 5.538626
iteration 300 / 1500: loss 2.307317
iteration 400 / 1500: loss 2.135517
iteration 500 / 1500: loss 2.136133
iteration 600 / 1500: loss 2.110058
iteration 700 / 1500: loss 2.125631
iteration 800 / 1500: loss 2.129607
iteration 900 / 1500: loss 2.081190
iteration 1000 / 1500: loss 2.146303
iteration 1100 / 1500: loss 2.117414
iteration 1200 / 1500: loss 2.097330
iteration 1300 / 1500: loss 2.141065
iteration 1400 / 1500: loss 2.142952
iteration 0 / 1500: loss 466.605378
iteration 100 / 1500: loss 84.824101
iteration 200 / 1500: loss 16.899667
iteration 300 / 1500: loss 4.711321
iteration 400 / 1500: loss 2.552826
iteration 500 / 1500: loss 2.180751
iteration 600 / 1500: loss 2.083772
iteration 700 / 1500: loss 2.034823
iteration 800 / 1500: loss 2.013730
iteration 900 / 1500: loss 2.036434
iteration 1000 / 1500: loss 2.070264
iteration 1100 / 1500: loss 2.023441
iteration 1200 / 1500: loss 2.073898
iteration 1300 / 1500: loss 2.057072
iteration 1400 / 1500: loss 2.022451
iteration 0 / 1500: loss 551.852837
iteration 100 / 1500: loss 73.237892
iteration 200 / 1500: loss 11.328216
iteration 300 / 1500: loss 3.272236
iteration 400 / 1500: loss 2.241274
```

```
iteration 500 / 1500: loss 2.088314
iteration 600 / 1500: loss 2.032625
iteration 700 / 1500: loss 2.003453
iteration 800 / 1500: loss 2.024816
iteration 900 / 1500: loss 2.030365
iteration 1000 / 1500: loss 2.155010
iteration 1100 / 1500: loss 2.074692
iteration 1200 / 1500: loss 2.070903
iteration 1300 / 1500: loss 2.092940
iteration 1400 / 1500: loss 2.062465
iteration 0 / 1500: loss 645.739858
iteration 100 / 1500: loss 62.671966
iteration 200 / 1500: loss 7.808051
iteration 300 / 1500: loss 2.585845
iteration 400 / 1500: loss 2.123233
iteration 500 / 1500: loss 2.071627
iteration 600 / 1500: loss 2.065274
iteration 700 / 1500: loss 2.075713
iteration 800 / 1500: loss 1.977609
iteration 900 / 1500: loss 2.048770
iteration 1000 / 1500: loss 2.079371
iteration 1100 / 1500: loss 2.074524
iteration 1200 / 1500: loss 2.125186
iteration 1300 / 1500: loss 2.066625
iteration 1400 / 1500: loss 2.062682
iteration 0 / 1500: loss 723.881707
iteration 100 / 1500: loss 51.476759
iteration 200 / 1500: loss 5.493343
iteration 300 / 1500: loss 2.327577
iteration 400 / 1500: loss 2.097159
iteration 500 / 1500: loss 2.069976
iteration 600 / 1500: loss 2.145041
iteration 700 / 1500: loss 2.116711
iteration 800 / 1500: loss 2.065275
iteration 900 / 1500: loss 2.087515
iteration 1000 / 1500: loss 2.159860
iteration 1100 / 1500: loss 2.041108
iteration 1200 / 1500: loss 2.117341
iteration 1300 / 1500: loss 2.072963
iteration 1400 / 1500: loss 2.088067
iteration 0 / 1500: loss 808.678891
iteration 100 / 1500: loss 42.181050
iteration 200 / 1500: loss 4.058780
iteration 300 / 1500: loss 2.226833
iteration 400 / 1500: loss 2.076042
iteration 500 / 1500: loss 2.046919
iteration 600 / 1500: loss 2.114088
iteration 700 / 1500: loss 2.133344
iteration 800 / 1500: loss 2.056423
iteration 900 / 1500: loss 2.080433
iteration 1000 / 1500: loss 2.111502
iteration 1100 / 1500: loss 2.124697
iteration 1200 / 1500: loss 2.066333
iteration 1300 / 1500: loss 2.039612
iteration 1400 / 1500: loss 2.162659
iteration 0 / 1500: loss 901.355422
iteration 100 / 1500: loss 34.472150
iteration 200 / 1500: loss 3.223862
iteration 300 / 1500: loss 2.187655
iteration 400 / 1500: loss 2.137672
iteration 500 / 1500: loss 2.110678
iteration 600 / 1500: loss 2.040873
iteration 700 / 1500: loss 2.100038
iteration 800 / 1500: loss 2.119612
```

```
iteration 900 / 1500: loss 2.109943
iteration 1000 / 1500: loss 2.134790
iteration 1100 / 1500: loss 2.086122
iteration 1200 / 1500: loss 2.150172
iteration 1300 / 1500: loss 2.085077
iteration 1400 / 1500: loss 2.158026
iteration 0 / 1500: loss 973.929409
iteration 100 / 1500: loss 27.434714
iteration 200 / 1500: loss 2.784409
iteration 300 / 1500: loss 2.153314
iteration 400 / 1500: loss 2.160903
iteration 500 / 1500: loss 2.110169
iteration 600 / 1500: loss 2.140008
iteration 700 / 1500: loss 2.097607
iteration 800 / 1500: loss 2.089068
iteration 900 / 1500: loss 2.084038
iteration 1000 / 1500: loss 2.113159
iteration 1100 / 1500: loss 2.128630
iteration 1200 / 1500: loss 2.111403
iteration 1300 / 1500: loss 2.107323
iteration 1400 / 1500: loss 2.082243
iteration 0 / 1500: loss 1063.725929
iteration 100 / 1500: loss 22.211027
iteration 200 / 1500: loss 2.502297
iteration 300 / 1500: loss 2.074317
iteration 400 / 1500: loss 2.105855
iteration 500 / 1500: loss 2.130633
iteration 600 / 1500: loss 2.128585
iteration 700 / 1500: loss 2.068602
iteration 800 / 1500: loss 2.159403
iteration 900 / 1500: loss 2.116027
iteration 1000 / 1500: loss 2.086126
iteration 1100 / 1500: loss 2.119299
iteration 1200 / 1500: loss 2.125860
iteration 1300 / 1500: loss 2.101325
iteration 1400 / 1500: loss 2.174343
iteration 0 / 1500: loss 1151.444224
iteration 100 / 1500: loss 17.931671
iteration 200 / 1500: loss 2.327824
iteration 300 / 1500: loss 2.095557
iteration 400 / 1500: loss 2.114769
iteration 500 / 1500: loss 2.166676
iteration 600 / 1500: loss 2.068356
iteration 700 / 1500: loss 2.139924
iteration 800 / 1500: loss 2.142332
iteration 900 / 1500: loss 2.150595
iteration 1000 / 1500: loss 2.130873
iteration 1100 / 1500: loss 2.049766
iteration 1200 / 1500: loss 2.131485
iteration 1300 / 1500: loss 2.117544
iteration 1400 / 1500: loss 2.097603
iteration 0 / 1500: loss 1230.929798
iteration 100 / 1500: loss 14.278493
iteration 200 / 1500: loss 2.257809
iteration 300 / 1500: loss 2.134355
iteration 400 / 1500: loss 2.174750
iteration 500 / 1500: loss 2.199089
iteration 600 / 1500: loss 2.128814
iteration 700 / 1500: loss 2.121965
iteration 800 / 1500: loss 2.103733
iteration 900 / 1500: loss 2.138032
iteration 1000 / 1500: loss 2.095716
iteration 1100 / 1500: loss 2.138523
iteration 1200 / 1500: loss 2.102196
```

```
iteration 1300 / 1500: loss 2.168366
iteration 1400 / 1500: loss 2.049158
iteration 0 / 1500: loss 470.937590
iteration 100 / 1500: loss 46.928237
iteration 200 / 1500: loss 6.396279
iteration 300 / 1500: loss 2.455553
iteration 400 / 1500: loss 2.119907
iteration 500 / 1500: loss 1.952391
iteration 600 / 1500: loss 2.022013
iteration 700 / 1500: loss 2.010234
iteration 800 / 1500: loss 2.094656
iteration 900 / 1500: loss 2.035679
iteration 1000 / 1500: loss 2.021630
iteration 1100 / 1500: loss 2.071714
iteration 1200 / 1500: loss 2.024700
iteration 1300 / 1500: loss 2.136598
iteration 1400 / 1500: loss 2.007162
iteration 0 / 1500: loss 549.876047
iteration 100 / 1500: loss 36.002793
iteration 200 / 1500: loss 4.203977
iteration 300 / 1500: loss 2.190259
iteration 400 / 1500: loss 2.026568
iteration 500 / 1500: loss 2.038096
iteration 600 / 1500: loss 2.055346
iteration 700 / 1500: loss 2.023571
iteration 800 / 1500: loss 2.097440
iteration 900 / 1500: loss 2.101311
iteration 1000 / 1500: loss 2.095096
iteration 1100 / 1500: loss 2.068668
iteration 1200 / 1500: loss 2.065320
iteration 1300 / 1500: loss 2.025109
iteration 1400 / 1500: loss 2.029344
iteration 0 / 1500: loss 633.702398
iteration 100 / 1500: loss 27.413328
iteration 200 / 1500: loss 3.122096
iteration 300 / 1500: loss 2.117413
iteration 400 / 1500: loss 2.061650
iteration 500 / 1500: loss 2.070752
iteration 600 / 1500: loss 2.066296
iteration 700 / 1500: loss 2.050559
iteration 800 / 1500: loss 2.088252
iteration 900 / 1500: loss 2.093144
iteration 1000 / 1500: loss 2.070022
iteration 1100 / 1500: loss 2.006099
iteration 1200 / 1500: loss 2.090541
iteration 1300 / 1500: loss 2.111553
iteration 1400 / 1500: loss 2.095188
iteration 0 / 1500: loss 723.203197
iteration 100 / 1500: loss 20.760593
iteration 200 / 1500: loss 2.551764
iteration 300 / 1500: loss 2.128096
iteration 400 / 1500: loss 2.105540
iteration 500 / 1500: loss 2.054644
iteration 600 / 1500: loss 2.069411
iteration 700 / 1500: loss 2.082874
iteration 800 / 1500: loss 2.090700
iteration 900 / 1500: loss 2.169140
iteration 1000 / 1500: loss 2.106842
iteration 1100 / 1500: loss 2.076365
iteration 1200 / 1500: loss 2.085967
iteration 1300 / 1500: loss 2.099697
iteration 1400 / 1500: loss 2.031922
iteration 0 / 1500: loss 801.567468
iteration 100 / 1500: loss 15.475888
```

```
iteration 200 / 1500: loss 2.268293
iteration 300 / 1500: loss 2.052393
iteration 400 / 1500: loss 2.054843
iteration 500 / 1500: loss 2.064917
iteration 600 / 1500: loss 2.047728
iteration 700 / 1500: loss 2.127098
iteration 800 / 1500: loss 2.094568
iteration 900 / 1500: loss 2.106266
iteration 1000 / 1500: loss 2.134668
iteration 1100 / 1500: loss 2.119554
iteration 1200 / 1500: loss 2.073591
iteration 1300 / 1500: loss 2.102609
iteration 1400 / 1500: loss 2.168354
iteration 0 / 1500: loss 889.786837
iteration 100 / 1500: loss 11.582050
iteration 200 / 1500: loss 2.262382
iteration 300 / 1500: loss 2.116021
iteration 400 / 1500: loss 2.107098
iteration 500 / 1500: loss 2.106341
iteration 600 / 1500: loss 2.019512
iteration 700 / 1500: loss 2.082053
iteration 800 / 1500: loss 2.095711
iteration 900 / 1500: loss 2.043065
iteration 1000 / 1500: loss 2.124887
iteration 1100 / 1500: loss 2.156159
iteration 1200 / 1500: loss 2.078683
iteration 1300 / 1500: loss 2.103950
iteration 1400 / 1500: loss 2.086201
iteration 0 / 1500: loss 979.405938
iteration 100 / 1500: loss 8.852340
iteration 200 / 1500: loss 2.222464
iteration 300 / 1500: loss 2.153625
iteration 400 / 1500: loss 2.138063
iteration 500 / 1500: loss 2.099598
iteration 600 / 1500: loss 2.073593
iteration 700 / 1500: loss 2.102728
iteration 800 / 1500: loss 2.130253
iteration 900 / 1500: loss 2.120526
iteration 1000 / 1500: loss 2.087761
iteration 1100 / 1500: loss 2.123362
iteration 1200 / 1500: loss 2.131821
iteration 1300 / 1500: loss 2.115773
iteration 1400 / 1500: loss 2.136262
iteration 0 / 1500: loss 1067.966531
iteration 100 / 1500: loss 6.822230
iteration 200 / 1500: loss 2.164982
iteration 300 / 1500: loss 2.179167
iteration 400 / 1500: loss 2.155326
iteration 500 / 1500: loss 2.124321
iteration 600 / 1500: loss 2.152417
iteration 700 / 1500: loss 2.117997
iteration 800 / 1500: loss 2.098044
iteration 900 / 1500: loss 2.090146
iteration 1000 / 1500: loss 2.058542
iteration 1100 / 1500: loss 2.164527
iteration 1200 / 1500: loss 2.138257
iteration 1300 / 1500: loss 2.139359
iteration 1400 / 1500: loss 2.145050
iteration 0 / 1500: loss 1147.214180
iteration 100 / 1500: loss 5.402835
iteration 200 / 1500: loss 2.131240
iteration 300 / 1500: loss 2.138318
iteration 400 / 1500: loss 2.103684
iteration 500 / 1500: loss 2.137866
```

```
iteration 600 / 1500: loss 2.139478
iteration 700 / 1500: loss 2.077535
iteration 800 / 1500: loss 2.113454
iteration 900 / 1500: loss 2.138580
iteration 1000 / 1500: loss 2.142717
iteration 1100 / 1500: loss 2.101264
iteration 1200 / 1500: loss 2.204633
iteration 1300 / 1500: loss 2.157272
iteration 1400 / 1500: loss 2.143970
iteration 0 / 1500: loss 1234.120468
iteration 100 / 1500: loss 4.386650
iteration 200 / 1500: loss 2.125198
iteration 300 / 1500: loss 2.133302
iteration 400 / 1500: loss 2.134907
iteration 500 / 1500: loss 2.121614
iteration 600 / 1500: loss 2.136522
iteration 700 / 1500: loss 2.174792
iteration 800 / 1500: loss 2.085738
iteration 900 / 1500: loss 2.107199
iteration 1000 / 1500: loss 2.126449
iteration 1100 / 1500: loss 2.142271
iteration 1200 / 1500: loss 2.148749
iteration 1300 / 1500: loss 2.110665
iteration 1400 / 1500: loss 2.111681
iteration 0 / 1500: loss 470.354794
iteration 100 / 1500: loss 26.097275
iteration 200 / 1500: loss 3.262598
iteration 300 / 1500: loss 2.137639
iteration 400 / 1500: loss 2.036700
iteration 500 / 1500: loss 2.119227
iteration 600 / 1500: loss 2.086207
iteration 700 / 1500: loss 2.031961
iteration 800 / 1500: loss 1.952520
iteration 900 / 1500: loss 2.057238
iteration 1000 / 1500: loss 2.088370
iteration 1100 / 1500: loss 2.063350
iteration 1200 / 1500: loss 2.043453
iteration 1300 / 1500: loss 2.020616
iteration 1400 / 1500: loss 1.991751
iteration 0 / 1500: loss 559.501608
iteration 100 / 1500: loss 18.523250
iteration 200 / 1500: loss 2.478435
iteration 300 / 1500: loss 2.150219
iteration 400 / 1500: loss 2.102557
iteration 500 / 1500: loss 2.021145
iteration 600 / 1500: loss 2.080256
iteration 700 / 1500: loss 2.071862
iteration 800 / 1500: loss 2.034569
iteration 900 / 1500: loss 1.978428
iteration 1000 / 1500: loss 2.081931
iteration 1100 / 1500: loss 2.099706
iteration 1200 / 1500: loss 2.036787
iteration 1300 / 1500: loss 2.064758
iteration 1400 / 1500: loss 2.078307
iteration 0 / 1500: loss 634.755284
iteration 100 / 1500: loss 12.792922
iteration 200 / 1500: loss 2.288672
iteration 300 / 1500: loss 2.073343
iteration 400 / 1500: loss 2.037305
iteration 500 / 1500: loss 2.038921
iteration 600 / 1500: loss 2.071574
iteration 700 / 1500: loss 2.094474
iteration 800 / 1500: loss 2.027029
iteration 900 / 1500: loss 2.046368
```

```
iteration 1000 / 1500: loss 2.070296
iteration 1100 / 1500: loss 2.108069
iteration 1200 / 1500: loss 2.060853
iteration 1300 / 1500: loss 2.136985
iteration 1400 / 1500: loss 2.071599
iteration 0 / 1500: loss 728.961049
iteration 100 / 1500: loss 9.123972
iteration 200 / 1500: loss 2.166182
iteration 300 / 1500: loss 2.089785
iteration 400 / 1500: loss 2.153832
iteration 500 / 1500: loss 2.043142
iteration 600 / 1500: loss 2.025890
iteration 700 / 1500: loss 2.073595
iteration 800 / 1500: loss 2.068082
iteration 900 / 1500: loss 2.107864
iteration 1000 / 1500: loss 2.091074
iteration 1100 / 1500: loss 2.018268
iteration 1200 / 1500: loss 2.067139
iteration 1300 / 1500: loss 2.035774
iteration 1400 / 1500: loss 2.117882
iteration 0 / 1500: loss 810.456402
iteration 100 / 1500: loss 6.579096
iteration 200 / 1500: loss 2.115090
iteration 300 / 1500: loss 2.121486
iteration 400 / 1500: loss 2.141285
iteration 500 / 1500: loss 2.084815
iteration 600 / 1500: loss 2.145284
iteration 700 / 1500: loss 2.026514
iteration 800 / 1500: loss 2.131659
iteration 900 / 1500: loss 2.033852
iteration 1000 / 1500: loss 2.189873
iteration 1100 / 1500: loss 2.067375
iteration 1200 / 1500: loss 2.169321
iteration 1300 / 1500: loss 2.067409
iteration 1400 / 1500: loss 2.055519
iteration 0 / 1500: loss 894.486644
iteration 100 / 1500: loss 4.975341
iteration 200 / 1500: loss 2.108216
iteration 300 / 1500: loss 2.093042
iteration 400 / 1500: loss 2.180978
iteration 500 / 1500: loss 2.144893
iteration 600 / 1500: loss 2.036163
iteration 700 / 1500: loss 2.088981
iteration 800 / 1500: loss 2.150793
iteration 900 / 1500: loss 2.112445
iteration 1000 / 1500: loss 2.127795
iteration 1100 / 1500: loss 2.110780
iteration 1200 / 1500: loss 2.137803
iteration 1300 / 1500: loss 2.099983
iteration 1400 / 1500: loss 2.047363
iteration 0 / 1500: loss 986.558341
iteration 100 / 1500: loss 3.934446
iteration 200 / 1500: loss 2.087829
iteration 300 / 1500: loss 2.183311
iteration 400 / 1500: loss 2.072661
iteration 500 / 1500: loss 2.142291
iteration 600 / 1500: loss 2.066415
iteration 700 / 1500: loss 2.103836
iteration 800 / 1500: loss 2.124405
iteration 900 / 1500: loss 2.091375
iteration 1000 / 1500: loss 2.142421
iteration 1100 / 1500: loss 2.099966
iteration 1200 / 1500: loss 2.042697
iteration 1300 / 1500: loss 2.070169
```

```
iteration 1400 / 1500: loss 2.067503
iteration 0 / 1500: loss 1064.759469
iteration 100 / 1500: loss 3.241849
iteration 200 / 1500: loss 2.151433
iteration 300 / 1500: loss 2.175917
iteration 400 / 1500: loss 2.129430
iteration 500 / 1500: loss 2.127096
iteration 600 / 1500: loss 2.156649
iteration 700 / 1500: loss 2.115737
iteration 800 / 1500: loss 2.133424
iteration 900 / 1500: loss 2.153045
iteration 1000 / 1500: loss 2.092976
iteration 1100 / 1500: loss 2.106639
iteration 1200 / 1500: loss 2.084222
iteration 1300 / 1500: loss 2.102945
iteration 1400 / 1500: loss 2.096861
iteration 0 / 1500: loss 1139.026068
iteration 100 / 1500: loss 2.789742
iteration 200 / 1500: loss 2.103307
iteration 300 / 1500: loss 2.110024
iteration 400 / 1500: loss 2.155524
iteration 500 / 1500: loss 2.135736
iteration 600 / 1500: loss 2.134397
iteration 700 / 1500: loss 2.132551
iteration 800 / 1500: loss 2.130794
iteration 900 / 1500: loss 2.071382
iteration 1000 / 1500: loss 2.167265
iteration 1100 / 1500: loss 2.166259
iteration 1200 / 1500: loss 2.135634
iteration 1300 / 1500: loss 2.083978
iteration 1400 / 1500: loss 2.152584
iteration 0 / 1500: loss 1231.298877
iteration 100 / 1500: loss 2.547612
iteration 200 / 1500: loss 2.169152
iteration 300 / 1500: loss 2.111401
iteration 400 / 1500: loss 2.159172
iteration 500 / 1500: loss 2.109042
iteration 600 / 1500: loss 2.122747
iteration 700 / 1500: loss 2.130441
iteration 800 / 1500: loss 2.134487
iteration 900 / 1500: loss 2.108888
iteration 1000 / 1500: loss 2.137583
iteration 1100 / 1500: loss 2.109839
iteration 1200 / 1500: loss 2.053532
iteration 1300 / 1500: loss 2.175289
iteration 1400 / 1500: loss 2.131541
iteration 0 / 1500: loss 472.543395
iteration 100 / 1500: loss 14.966085
iteration 200 / 1500: loss 2.398877
iteration 300 / 1500: loss 2.026123
iteration 400 / 1500: loss 2.031774
iteration 500 / 1500: loss 2.053824
iteration 600 / 1500: loss 2.058594
iteration 700 / 1500: loss 2.043284
iteration 800 / 1500: loss 2.034948
iteration 900 / 1500: loss 1.964046
iteration 1000 / 1500: loss 2.058231
iteration 1100 / 1500: loss 1.997258
iteration 1200 / 1500: loss 2.070701
iteration 1300 / 1500: loss 2.043830
iteration 1400 / 1500: loss 2.048102
iteration 0 / 1500: loss 549.596235
iteration 100 / 1500: loss 9.811628
iteration 200 / 1500: loss 2.147948
```

```
iteration 300 / 1500: loss 2.097004
iteration 400 / 1500: loss 2.004228
iteration 500 / 1500: loss 2.069584
iteration 600 / 1500: loss 2.057639
iteration 700 / 1500: loss 2.077047
iteration 800 / 1500: loss 2.145105
iteration 900 / 1500: loss 2.039796
iteration 1000 / 1500: loss 2.086446
iteration 1100 / 1500: loss 1.990390
iteration 1200 / 1500: loss 2.053035
iteration 1300 / 1500: loss 2.015476
iteration 1400 / 1500: loss 2.003171
iteration 0 / 1500: loss 631.247524
iteration 100 / 1500: loss 6.592871
iteration 200 / 1500: loss 2.118717
iteration 300 / 1500: loss 2.112623
iteration 400 / 1500: loss 2.091887
iteration 500 / 1500: loss 1.973431
iteration 600 / 1500: loss 2.113087
iteration 700 / 1500: loss 2.068943
iteration 800 / 1500: loss 2.150161
iteration 900 / 1500: loss 2.040559
iteration 1000 / 1500: loss 2.077525
iteration 1100 / 1500: loss 2.053386
iteration 1200 / 1500: loss 2.051920
iteration 1300 / 1500: loss 2.099409
iteration 1400 / 1500: loss 2.070228
iteration 0 / 1500: loss 721.905721
iteration 100 / 1500: loss 4.702694
iteration 200 / 1500: loss 2.098145
iteration 300 / 1500: loss 2.055403
iteration 400 / 1500: loss 2.069431
iteration 500 / 1500: loss 2.054201
iteration 600 / 1500: loss 2.066326
iteration 700 / 1500: loss 2.040197
iteration 800 / 1500: loss 2.075309
iteration 900 / 1500: loss 2.132689
iteration 1000 / 1500: loss 2.075635
iteration 1100 / 1500: loss 2.128276
iteration 1200 / 1500: loss 2.083616
iteration 1300 / 1500: loss 2.085297
iteration 1400 / 1500: loss 2.081638
iteration 0 / 1500: loss 807.086432
iteration 100 / 1500: loss 3.602589
iteration 200 / 1500: loss 2.064727
iteration 300 / 1500: loss 2.066757
iteration 400 / 1500: loss 2.161149
iteration 500 / 1500: loss 2.112810
iteration 600 / 1500: loss 2.046946
iteration 700 / 1500: loss 2.106004
iteration 800 / 1500: loss 2.074399
iteration 900 / 1500: loss 2.058606
iteration 1000 / 1500: loss 2.109823
iteration 1100 / 1500: loss 2.117083
iteration 1200 / 1500: loss 2.064522
iteration 1300 / 1500: loss 2.110526
iteration 1400 / 1500: loss 2.136187
iteration 0 / 1500: loss 881.997243
iteration 100 / 1500: loss 2.865939
iteration 200 / 1500: loss 2.099393
iteration 300 / 1500: loss 2.133588
iteration 400 / 1500: loss 2.102321
iteration 500 / 1500: loss 2.071595
iteration 600 / 1500: loss 2.100366
```

```
iteration 700 / 1500: loss 2.085835
iteration 800 / 1500: loss 2.175420
iteration 900 / 1500: loss 2.084882
iteration 1000 / 1500: loss 2.059804
iteration 1100 / 1500: loss 2.103409
iteration 1200 / 1500: loss 2.121930
iteration 1300 / 1500: loss 2.110456
iteration 1400 / 1500: loss 2.071308
iteration 0 / 1500: loss 981.461899
iteration 100 / 1500: loss 2.566610
iteration 200 / 1500: loss 2.135261
iteration 300 / 1500: loss 2.130266
iteration 400 / 1500: loss 2.147319
iteration 500 / 1500: loss 2.147331
iteration 600 / 1500: loss 2.132186
iteration 700 / 1500: loss 2.114448
iteration 800 / 1500: loss 2.190309
iteration 900 / 1500: loss 2.038506
iteration 1000 / 1500: loss 2.114557
iteration 1100 / 1500: loss 2.061074
iteration 1200 / 1500: loss 2.115368
iteration 1300 / 1500: loss 2.076857
iteration 1400 / 1500: loss 2.087160
iteration 0 / 1500: loss 1068.643029
iteration 100 / 1500: loss 2.402419
iteration 200 / 1500: loss 2.151826
iteration 300 / 1500: loss 2.093278
iteration 400 / 1500: loss 2.106104
iteration 500 / 1500: loss 2.120589
iteration 600 / 1500: loss 2.142718
iteration 700 / 1500: loss 2.138441
iteration 800 / 1500: loss 2.102089
iteration 900 / 1500: loss 2.050639
iteration 1000 / 1500: loss 2.165156
iteration 1100 / 1500: loss 2.148135
iteration 1200 / 1500: loss 2.083546
iteration 1300 / 1500: loss 2.115831
iteration 1400 / 1500: loss 2.168561
iteration 0 / 1500: loss 1142.110353
iteration 100 / 1500: loss 2.233587
iteration 200 / 1500: loss 2.115142
iteration 300 / 1500: loss 2.122041
iteration 400 / 1500: loss 2.126315
iteration 500 / 1500: loss 2.097549
iteration 600 / 1500: loss 2.115076
iteration 700 / 1500: loss 2.143004
iteration 800 / 1500: loss 2.068077
iteration 900 / 1500: loss 2.129132
iteration 1000 / 1500: loss 2.151559
iteration 1100 / 1500: loss 2.088708
iteration 1200 / 1500: loss 2.114439
iteration 1300 / 1500: loss 2.099329
iteration 1400 / 1500: loss 2.100582
iteration 0 / 1500: loss 1240.611650
iteration 100 / 1500: loss 2.257224
iteration 200 / 1500: loss 2.142894
iteration 300 / 1500: loss 2.145502
iteration 400 / 1500: loss 2.109090
iteration 500 / 1500: loss 2.174177
iteration 600 / 1500: loss 2.135544
iteration 700 / 1500: loss 2.130582
iteration 800 / 1500: loss 2.133300
iteration 900 / 1500: loss 2.151190
iteration 1000 / 1500: loss 2.184308
```

```
iteration 1100 / 1500: loss 2.127242
iteration 1200 / 1500: loss 2.125593
iteration 1300 / 1500: loss 2.125140
iteration 1400 / 1500: loss 2.138874
iteration 0 / 1500: loss 466.105810
iteration 100 / 1500: loss 8.870576
iteration 200 / 1500: loss 2.184462
iteration 300 / 1500: loss 2.056609
iteration 400 / 1500: loss 2.069455
iteration 500 / 1500: loss 2.044986
iteration 600 / 1500: loss 1.983023
iteration 700 / 1500: loss 2.050641
iteration 800 / 1500: loss 2.082967
iteration 900 / 1500: loss 2.060668
iteration 1000 / 1500: loss 2.074452
iteration 1100 / 1500: loss 2.044638
iteration 1200 / 1500: loss 1.982562
iteration 1300 / 1500: loss 1.996934
iteration 1400 / 1500: loss 2.029978
iteration 0 / 1500: loss 554.937691
iteration 100 / 1500: loss 5.797895
iteration 200 / 1500: loss 2.163053
iteration 300 / 1500: loss 2.114743
iteration 400 / 1500: loss 2.049452
iteration 500 / 1500: loss 2.096755
iteration 600 / 1500: loss 2.065586
iteration 700 / 1500: loss 1.983859
iteration 800 / 1500: loss 2.102799
iteration 900 / 1500: loss 2.096293
iteration 1000 / 1500: loss 2.054803
iteration 1100 / 1500: loss 2.068833
iteration 1200 / 1500: loss 2.058812
iteration 1300 / 1500: loss 2.078124
iteration 1400 / 1500: loss 2.116251
iteration 0 / 1500: loss 636.692098
iteration 100 / 1500: loss 3.957086
iteration 200 / 1500: loss 2.109150
iteration 300 / 1500: loss 2.138302
iteration 400 / 1500: loss 2.023887
iteration 500 / 1500: loss 2.077563
iteration 600 / 1500: loss 2.040160
iteration 700 / 1500: loss 2.129149
iteration 800 / 1500: loss 2.083723
iteration 900 / 1500: loss 2.121839
iteration 1000 / 1500: loss 2.072442
iteration 1100 / 1500: loss 2.086640
iteration 1200 / 1500: loss 2.085641
iteration 1300 / 1500: loss 2.098394
iteration 1400 / 1500: loss 2.085959
iteration 0 / 1500: loss 716.705081
iteration 100 / 1500: loss 3.090659
iteration 200 / 1500: loss 2.089429
iteration 300 / 1500: loss 2.138248
iteration 400 / 1500: loss 2.050337
iteration 500 / 1500: loss 2.016383
iteration 600 / 1500: loss 2.147378
iteration 700 / 1500: loss 2.085735
iteration 800 / 1500: loss 2.135524
iteration 900 / 1500: loss 2.108157
iteration 1000 / 1500: loss 2.068910
iteration 1100 / 1500: loss 2.101350
iteration 1200 / 1500: loss 2.072654
iteration 1300 / 1500: loss 2.034145
iteration 1400 / 1500: loss 2.078297
```

```
iteration 0 / 1500: loss 814.167312
iteration 100 / 1500: loss 2.585511
iteration 200 / 1500: loss 2.163770
iteration 300 / 1500: loss 2.101267
iteration 400 / 1500: loss 2.084453
iteration 500 / 1500: loss 2.073999
iteration 600 / 1500: loss 2.109921
iteration 700 / 1500: loss 2.156321
iteration 800 / 1500: loss 2.093714
iteration 900 / 1500: loss 2.045200
iteration 1000 / 1500: loss 2.152159
iteration 1100 / 1500: loss 2.085583
iteration 1200 / 1500: loss 2.040063
iteration 1300 / 1500: loss 2.034567
iteration 1400 / 1500: loss 2.050659
iteration 0 / 1500: loss 902.792516
iteration 100 / 1500: loss 2.404453
iteration 200 / 1500: loss 2.081397
iteration 300 / 1500: loss 2.048022
iteration 400 / 1500: loss 2.123063
iteration 500 / 1500: loss 2.050313
iteration 600 / 1500: loss 2.108795
iteration 700 / 1500: loss 2.113340
iteration 800 / 1500: loss 2.147521
iteration 900 / 1500: loss 2.131647
iteration 1000 / 1500: loss 2.113658
iteration 1100 / 1500: loss 2.150340
iteration 1200 / 1500: loss 2.035591
iteration 1300 / 1500: loss 2.122850
iteration 1400 / 1500: loss 2.141562
iteration 0 / 1500: loss 967.997922
iteration 100 / 1500: loss 2.163913
iteration 200 / 1500: loss 2.077664
iteration 300 / 1500: loss 2.077956
iteration 400 / 1500: loss 2.148088
iteration 500 / 1500: loss 2.144002
iteration 600 / 1500: loss 2.067302
iteration 700 / 1500: loss 2.075572
iteration 800 / 1500: loss 2.108994
iteration 900 / 1500: loss 2.106906
iteration 1000 / 1500: loss 2.137693
iteration 1100 / 1500: loss 2.106768
iteration 1200 / 1500: loss 2.091516
iteration 1300 / 1500: loss 2.075396
iteration 1400 / 1500: loss 2.146994
iteration 0 / 1500: loss 1056.927835
iteration 100 / 1500: loss 2.119268
iteration 200 / 1500: loss 2.172836
iteration 300 / 1500: loss 2.132397
iteration 400 / 1500: loss 2.138547
iteration 500 / 1500: loss 2.059640
iteration 600 / 1500: loss 2.150276
iteration 700 / 1500: loss 2.119207
iteration 800 / 1500: loss 2.095907
iteration 900 / 1500: loss 2.082741
iteration 1000 / 1500: loss 2.043232
iteration 1100 / 1500: loss 2.114741
iteration 1200 / 1500: loss 2.108964
iteration 1300 / 1500: loss 2.152511
iteration 1400 / 1500: loss 2.155311
iteration 0 / 1500: loss 1138.313579
iteration 100 / 1500: loss 2.165891
iteration 200 / 1500: loss 2.165059
iteration 300 / 1500: loss 2.112433
```

```
iteration 400 / 1500: loss 2.167402
iteration 500 / 1500: loss 2.144708
iteration 600 / 1500: loss 2.121226
iteration 700 / 1500: loss 2.122362
iteration 800 / 1500: loss 2.111661
iteration 900 / 1500: loss 2.116857
iteration 1000 / 1500: loss 2.098448
iteration 1100 / 1500: loss 2.098843
iteration 1200 / 1500: loss 2.093621
iteration 1300 / 1500: loss 2.164110
iteration 1400 / 1500: loss 2.049766
iteration 0 / 1500: loss 1235.025682
iteration 100 / 1500: loss 2.172116
iteration 200 / 1500: loss 2.063853
iteration 300 / 1500: loss 2.176245
iteration 400 / 1500: loss 2.158220
iteration 500 / 1500: loss 2.108315
iteration 600 / 1500: loss 2.118096
iteration 700 / 1500: loss 2.145181
iteration 800 / 1500: loss 2.092816
iteration 900 / 1500: loss 2.120452
iteration 1000 / 1500: loss 2.127843
iteration 1100 / 1500: loss 2.076002
iteration 1200 / 1500: loss 2.102984
iteration 1300 / 1500: loss 2.136623
iteration 1400 / 1500: loss 2.127914
iteration 0 / 1500: loss 467.266543
iteration 100 / 1500: loss 5.694995
iteration 200 / 1500: loss 2.088360
iteration 300 / 1500: loss 2.024558
iteration 400 / 1500: loss 2.056793
iteration 500 / 1500: loss 2.009103
iteration 600 / 1500: loss 2.091880
iteration 700 / 1500: loss 2.041354
iteration 800 / 1500: loss 2.029119
iteration 900 / 1500: loss 2.077337
iteration 1000 / 1500: loss 2.098785
iteration 1100 / 1500: loss 2.005964
iteration 1200 / 1500: loss 2.087275
iteration 1300 / 1500: loss 2.072524
iteration 1400 / 1500: loss 2.071917
iteration 0 / 1500: loss 553.759278
iteration 100 / 1500: loss 3.805606
iteration 200 / 1500: loss 2.071664
iteration 300 / 1500: loss 2.031119
iteration 400 / 1500: loss 2.074871
iteration 500 / 1500: loss 2.073819
iteration 600 / 1500: loss 2.096404
iteration 700 / 1500: loss 2.089691
iteration 800 / 1500: loss 2.136330
iteration 900 / 1500: loss 2.025471
iteration 1000 / 1500: loss 2.041067
iteration 1100 / 1500: loss 2.100591
iteration 1200 / 1500: loss 1.982328
iteration 1300 / 1500: loss 2.073711
iteration 1400 / 1500: loss 2.055383
iteration 0 / 1500: loss 639.738322
iteration 100 / 1500: loss 2.894104
iteration 200 / 1500: loss 2.042771
iteration 300 / 1500: loss 2.038379
iteration 400 / 1500: loss 2.096334
iteration 500 / 1500: loss 2.077909
iteration 600 / 1500: loss 2.096564
iteration 700 / 1500: loss 2.065873
```

```
iteration 800 / 1500: loss 2.110165
iteration 900 / 1500: loss 2.120116
iteration 1000 / 1500: loss 2.085650
iteration 1100 / 1500: loss 2.054848
iteration 1200 / 1500: loss 2.180718
iteration 1300 / 1500: loss 2.084374
iteration 1400 / 1500: loss 2.048631
iteration 0 / 1500: loss 719.921881
iteration 100 / 1500: loss 2.501034
iteration 200 / 1500: loss 2.093003
iteration 300 / 1500: loss 2.049918
iteration 400 / 1500: loss 2.124167
iteration 500 / 1500: loss 2.092283
iteration 600 / 1500: loss 2.130371
iteration 700 / 1500: loss 2.168349
iteration 800 / 1500: loss 2.146460
iteration 900 / 1500: loss 2.118611
iteration 1000 / 1500: loss 2.081847
iteration 1100 / 1500: loss 2.075311
iteration 1200 / 1500: loss 2.113794
iteration 1300 / 1500: loss 2.107447
iteration 1400 / 1500: loss 2.110096
iteration 0 / 1500: loss 815.432720
iteration 100 / 1500: loss 2.277290
iteration 200 / 1500: loss 2.075281
iteration 300 / 1500: loss 2.045595
iteration 400 / 1500: loss 2.145175
iteration 500 / 1500: loss 2.050841
iteration 600 / 1500: loss 2.101286
iteration 700 / 1500: loss 2.117558
iteration 800 / 1500: loss 2.120214
iteration 900 / 1500: loss 2.102350
iteration 1000 / 1500: loss 2.075543
iteration 1100 / 1500: loss 2.122979
iteration 1200 / 1500: loss 2.056145
iteration 1300 / 1500: loss 2.076212
iteration 1400 / 1500: loss 2.125302
iteration 0 / 1500: loss 891.099939
iteration 100 / 1500: loss 2.226085
iteration 200 / 1500: loss 2.123615
iteration 300 / 1500: loss 2.129464
iteration 400 / 1500: loss 2.079126
iteration 500 / 1500: loss 2.064320
iteration 600 / 1500: loss 2.093638
iteration 700 / 1500: loss 2.153648
iteration 800 / 1500: loss 2.150737
iteration 900 / 1500: loss 2.101615
iteration 1000 / 1500: loss 2.067045
iteration 1100 / 1500: loss 2.107193
iteration 1200 / 1500: loss 2.141119
iteration 1300 / 1500: loss 2.107476
iteration 1400 / 1500: loss 2.107766
iteration 0 / 1500: loss 975.160127
iteration 100 / 1500: loss 2.170841
iteration 200 / 1500: loss 2.081013
iteration 300 / 1500: loss 2.082717
iteration 400 / 1500: loss 2.093519
iteration 500 / 1500: loss 2.216109
iteration 600 / 1500: loss 2.114383
iteration 700 / 1500: loss 2.114413
iteration 800 / 1500: loss 2.135828
iteration 900 / 1500: loss 2.151438
iteration 1000 / 1500: loss 2.085303
iteration 1100 / 1500: loss 2.120017
```

```
iteration 1200 / 1500: loss 2.109237
iteration 1300 / 1500: loss 2.113717
iteration 1400 / 1500: loss 2.107258
iteration 0 / 1500: loss 1059.491903
iteration 100 / 1500: loss 2.126063
iteration 200 / 1500: loss 2.122128
iteration 300 / 1500: loss 2.093304
iteration 400 / 1500: loss 2.125842
iteration 500 / 1500: loss 2.095607
iteration 600 / 1500: loss 2.108787
iteration 700 / 1500: loss 2.088020
iteration 800 / 1500: loss 2.152114
iteration 900 / 1500: loss 2.092485
iteration 1000 / 1500: loss 2.123666
iteration 1100 / 1500: loss 2.179786
iteration 1200 / 1500: loss 2.178171
iteration 1300 / 1500: loss 2.087428
iteration 1400 / 1500: loss 2.168152
iteration 0 / 1500: loss 1162.859253
iteration 100 / 1500: loss 2.171716
iteration 200 / 1500: loss 2.162720
iteration 300 / 1500: loss 2.159415
iteration 400 / 1500: loss 2.097170
iteration 500 / 1500: loss 2.115323
iteration 600 / 1500: loss 2.074130
iteration 700 / 1500: loss 2.088638
iteration 800 / 1500: loss 2.064163
iteration 900 / 1500: loss 2.169872
iteration 1000 / 1500: loss 2.143507
iteration 1100 / 1500: loss 2.123922
iteration 1200 / 1500: loss 2.180886
iteration 1300 / 1500: loss 2.107531
iteration 1400 / 1500: loss 2.170842
iteration 0 / 1500: loss 1246.716101
iteration 100 / 1500: loss 2.143575
iteration 200 / 1500: loss 2.118596
iteration 300 / 1500: loss 2.191422
iteration 400 / 1500: loss 2.079679
iteration 500 / 1500: loss 2.147612
iteration 600 / 1500: loss 2.123344
iteration 700 / 1500: loss 2.131276
iteration 800 / 1500: loss 2.133711
iteration 900 / 1500: loss 2.154185
iteration 1000 / 1500: loss 2.211470
iteration 1100 / 1500: loss 2.132813
iteration 1200 / 1500: loss 2.151343
iteration 1300 / 1500: loss 2.124164
iteration 1400 / 1500: loss 2.068624
iteration 0 / 1500: loss 471.642651
iteration 100 / 1500: loss 4.031040
iteration 200 / 1500: loss 2.029315
iteration 300 / 1500: loss 2.055035
iteration 400 / 1500: loss 2.017483
iteration 500 / 1500: loss 2.130123
iteration 600 / 1500: loss 2.028285
iteration 700 / 1500: loss 2.045799
iteration 800 / 1500: loss 1.995489
iteration 900 / 1500: loss 2.041419
iteration 1000 / 1500: loss 2.115482
iteration 1100 / 1500: loss 1.988181
iteration 1200 / 1500: loss 2.041211
iteration 1300 / 1500: loss 2.076127
iteration 1400 / 1500: loss 2.071525
iteration 0 / 1500: loss 556.663710
```

```
iteration 100 / 1500: loss 2.870107
iteration 200 / 1500: loss 2.084467
iteration 300 / 1500: loss 2.045323
iteration 400 / 1500: loss 2.094691
iteration 500 / 1500: loss 2.028877
iteration 600 / 1500: loss 2.055283
iteration 700 / 1500: loss 2.029147
iteration 800 / 1500: loss 2.041836
iteration 900 / 1500: loss 2.002079
iteration 1000 / 1500: loss 2.076038
iteration 1100 / 1500: loss 2.087174
iteration 1200 / 1500: loss 2.098564
iteration 1300 / 1500: loss 2.169864
iteration 1400 / 1500: loss 2.103360
iteration 0 / 1500: loss 634.779614
iteration 100 / 1500: loss 2.395575
iteration 200 / 1500: loss 2.104136
iteration 300 / 1500: loss 2.098565
iteration 400 / 1500: loss 2.078249
iteration 500 / 1500: loss 2.102406
iteration 600 / 1500: loss 2.104306
iteration 700 / 1500: loss 2.004028
iteration 800 / 1500: loss 2.074071
iteration 900 / 1500: loss 2.064732
iteration 1000 / 1500: loss 2.106805
iteration 1100 / 1500: loss 2.107380
iteration 1200 / 1500: loss 2.147017
iteration 1300 / 1500: loss 2.184200
iteration 1400 / 1500: loss 2.121810
iteration 0 / 1500: loss 717.993004
iteration 100 / 1500: loss 2.248431
iteration 200 / 1500: loss 2.092809
iteration 300 / 1500: loss 2.093111
iteration 400 / 1500: loss 2.101515
iteration 500 / 1500: loss 2.055965
iteration 600 / 1500: loss 2.087965
iteration 700 / 1500: loss 2.023518
iteration 800 / 1500: loss 2.064529
iteration 900 / 1500: loss 2.051211
iteration 1000 / 1500: loss 2.063640
iteration 1100 / 1500: loss 2.111157
iteration 1200 / 1500: loss 1.998179
iteration 1300 / 1500: loss 2.101337
iteration 1400 / 1500: loss 2.082031
iteration 0 / 1500: loss 804.810426
iteration 100 / 1500: loss 2.081846
iteration 200 / 1500: loss 2.055732
iteration 300 / 1500: loss 2.105090
iteration 400 / 1500: loss 2.099577
iteration 500 / 1500: loss 2.151538
iteration 600 / 1500: loss 2.048849
iteration 700 / 1500: loss 2.074820
iteration 800 / 1500: loss 2.070509
iteration 900 / 1500: loss 2.090750
iteration 1000 / 1500: loss 2.071036
iteration 1100 / 1500: loss 2.095904
iteration 1200 / 1500: loss 2.059259
iteration 1300 / 1500: loss 2.080722
iteration 1400 / 1500: loss 2.111693
iteration 0 / 1500: loss 892.538776
iteration 100 / 1500: loss 2.050543
iteration 200 / 1500: loss 2.122545
iteration 300 / 1500: loss 2.021466
iteration 400 / 1500: loss 2.116574
```

```
iteration 500 / 1500: loss 2.133235
iteration 600 / 1500: loss 2.185287
iteration 700 / 1500: loss 2.129973
iteration 800 / 1500: loss 2.049527
iteration 900 / 1500: loss 2.168705
iteration 1000 / 1500: loss 2.132791
iteration 1100 / 1500: loss 2.101919
iteration 1200 / 1500: loss 2.096535
iteration 1300 / 1500: loss 2.130244
iteration 1400 / 1500: loss 2.152279
iteration 0 / 1500: loss 978.421020
iteration 100 / 1500: loss 2.105735
iteration 200 / 1500: loss 2.092988
iteration 300 / 1500: loss 2.146775
iteration 400 / 1500: loss 2.123716
iteration 500 / 1500: loss 2.124691
iteration 600 / 1500: loss 2.057117
iteration 700 / 1500: loss 2.104821
iteration 800 / 1500: loss 2.129328
iteration 900 / 1500: loss 2.132307
iteration 1000 / 1500: loss 2.104280
iteration 1100 / 1500: loss 2.092184
iteration 1200 / 1500: loss 2.046483
iteration 1300 / 1500: loss 2.156946
iteration 1400 / 1500: loss 2.125359
iteration 0 / 1500: loss 1061.267737
iteration 100 / 1500: loss 2.150430
iteration 200 / 1500: loss 2.172350
iteration 300 / 1500: loss 2.148304
iteration 400 / 1500: loss 2.136129
iteration 500 / 1500: loss 2.141354
iteration 600 / 1500: loss 2.183301
iteration 700 / 1500: loss 2.141115
iteration 800 / 1500: loss 2.138731
iteration 900 / 1500: loss 2.115277
iteration 1000 / 1500: loss 2.093788
iteration 1100 / 1500: loss 2.098716
iteration 1200 / 1500: loss 2.130133
iteration 1300 / 1500: loss 2.134812
iteration 1400 / 1500: loss 2.093540
iteration 0 / 1500: loss 1153.714607
iteration 100 / 1500: loss 2.076982
iteration 200 / 1500: loss 2.190846
iteration 300 / 1500: loss 2.143713
iteration 400 / 1500: loss 2.106288
iteration 500 / 1500: loss 2.105102
iteration 600 / 1500: loss 2.134681
iteration 700 / 1500: loss 2.064435
iteration 800 / 1500: loss 2.129468
iteration 900 / 1500: loss 2.170095
iteration 1000 / 1500: loss 2.132957
iteration 1100 / 1500: loss 2.139649
iteration 1200 / 1500: loss 2.172389
iteration 1300 / 1500: loss 2.100929
iteration 1400 / 1500: loss 2.158942
iteration 0 / 1500: loss 1231.235252
iteration 100 / 1500: loss 2.093265
iteration 200 / 1500: loss 2.078359
iteration 300 / 1500: loss 2.168973
iteration 400 / 1500: loss 2.141261
iteration 500 / 1500: loss 2.139326
iteration 600 / 1500: loss 2.162487
iteration 700 / 1500: loss 2.077057
iteration 800 / 1500: loss 2.150929
```

```
iteration 900 / 1500: loss 2.113929
iteration 1000 / 1500: loss 2.158333
iteration 1100 / 1500: loss 2.123516
iteration 1200 / 1500: loss 2.147544
iteration 1300 / 1500: loss 2.090607
iteration 1400 / 1500: loss 2.150859
iteration 0 / 1500: loss 467.122126
iteration 100 / 1500: loss 3.146612
iteration 200 / 1500: loss 2.047183
iteration 300 / 1500: loss 2.126848
iteration 400 / 1500: loss 2.126112
iteration 500 / 1500: loss 2.108552
iteration 600 / 1500: loss 2.032445
iteration 700 / 1500: loss 2.019633
iteration 800 / 1500: loss 1.997095
iteration 900 / 1500: loss 2.040159
iteration 1000 / 1500: loss 2.065858
iteration 1100 / 1500: loss 2.055456
iteration 1200 / 1500: loss 2.073238
iteration 1300 / 1500: loss 2.012458
iteration 1400 / 1500: loss 2.058727
iteration 0 / 1500: loss 550.999632
iteration 100 / 1500: loss 2.441205
iteration 200 / 1500: loss 1.964674
iteration 300 / 1500: loss 2.093344
iteration 400 / 1500: loss 2.083172
iteration 500 / 1500: loss 2.065520
iteration 600 / 1500: loss 2.010713
iteration 700 / 1500: loss 2.112156
iteration 800 / 1500: loss 2.105747
iteration 900 / 1500: loss 2.063571
iteration 1000 / 1500: loss 2.097460
iteration 1100 / 1500: loss 2.096861
iteration 1200 / 1500: loss 2.057498
iteration 1300 / 1500: loss 2.067781
iteration 1400 / 1500: loss 2.037256
iteration 0 / 1500: loss 640.710956
iteration 100 / 1500: loss 2.238881
iteration 200 / 1500: loss 2.021092
iteration 300 / 1500: loss 2.098815
iteration 400 / 1500: loss 2.068379
iteration 500 / 1500: loss 2.071556
iteration 600 / 1500: loss 2.045587
iteration 700 / 1500: loss 2.056877
iteration 800 / 1500: loss 2.079690
iteration 900 / 1500: loss 2.082544
iteration 1000 / 1500: loss 2.101063
iteration 1100 / 1500: loss 2.109228
iteration 1200 / 1500: loss 2.099474
iteration 1300 / 1500: loss 2.077549
iteration 1400 / 1500: loss 2.094232
iteration 0 / 1500: loss 723.761099
iteration 100 / 1500: loss 2.102509
iteration 200 / 1500: loss 2.069982
iteration 300 / 1500: loss 2.048351
iteration 400 / 1500: loss 2.118834
iteration 500 / 1500: loss 2.069880
iteration 600 / 1500: loss 2.061332
iteration 700 / 1500: loss 2.134717
iteration 800 / 1500: loss 2.077454
iteration 900 / 1500: loss 2.066949
iteration 1000 / 1500: loss 2.118859
iteration 1100 / 1500: loss 2.054138
iteration 1200 / 1500: loss 2.062460
```

```
iteration 1300 / 1500: loss 2.083230
iteration 1400 / 1500: loss 2.077583
iteration 0 / 1500: loss 814.289697
iteration 100 / 1500: loss 2.144217
iteration 200 / 1500: loss 2.117028
iteration 300 / 1500: loss 2.071299
iteration 400 / 1500: loss 2.090707
iteration 500 / 1500: loss 2.082095
iteration 600 / 1500: loss 2.119077
iteration 700 / 1500: loss 2.131515
iteration 800 / 1500: loss 2.139522
iteration 900 / 1500: loss 2.071935
iteration 1000 / 1500: loss 1.995802
iteration 1100 / 1500: loss 2.080709
iteration 1200 / 1500: loss 2.069053
iteration 1300 / 1500: loss 2.158015
iteration 1400 / 1500: loss 2.013248
iteration 0 / 1500: loss 882.651028
iteration 100 / 1500: loss 2.087761
iteration 200 / 1500: loss 2.190231
iteration 300 / 1500: loss 2.049567
iteration 400 / 1500: loss 2.118111
iteration 500 / 1500: loss 2.105311
iteration 600 / 1500: loss 2.115002
iteration 700 / 1500: loss 2.098542
iteration 800 / 1500: loss 2.071331
iteration 900 / 1500: loss 2.159080
iteration 1000 / 1500: loss 2.090800
iteration 1100 / 1500: loss 2.115137
iteration 1200 / 1500: loss 2.078494
iteration 1300 / 1500: loss 2.162096
iteration 1400 / 1500: loss 2.068033
iteration 0 / 1500: loss 982.381256
iteration 100 / 1500: loss 2.070057
iteration 200 / 1500: loss 2.128213
iteration 300 / 1500: loss 2.126240
iteration 400 / 1500: loss 2.106075
iteration 500 / 1500: loss 2.145987
iteration 600 / 1500: loss 2.077072
iteration 700 / 1500: loss 2.100437
iteration 800 / 1500: loss 2.139017
iteration 900 / 1500: loss 2.100861
iteration 1000 / 1500: loss 2.104368
iteration 1100 / 1500: loss 2.126603
iteration 1200 / 1500: loss 2.147854
iteration 1300 / 1500: loss 2.092017
iteration 1400 / 1500: loss 2.090756
iteration 0 / 1500: loss 1061.798875
iteration 100 / 1500: loss 2.130780
iteration 200 / 1500: loss 2.086699
iteration 300 / 1500: loss 2.167264
iteration 400 / 1500: loss 2.105385
iteration 500 / 1500: loss 2.157068
iteration 600 / 1500: loss 2.211300
iteration 700 / 1500: loss 2.129535
iteration 800 / 1500: loss 2.143764
iteration 900 / 1500: loss 2.128213
iteration 1000 / 1500: loss 2.144176
iteration 1100 / 1500: loss 2.152060
iteration 1200 / 1500: loss 2.085546
iteration 1300 / 1500: loss 2.117069
iteration 1400 / 1500: loss 2.119684
iteration 0 / 1500: loss 1141.771078
iteration 100 / 1500: loss 2.142591
```

```
iteration 200 / 1500: loss 2.186089
iteration 300 / 1500: loss 2.117621
iteration 400 / 1500: loss 2.191261
iteration 500 / 1500: loss 2.074274
iteration 600 / 1500: loss 2.137498
iteration 700 / 1500: loss 2.092279
iteration 800 / 1500: loss 2.111192
iteration 900 / 1500: loss 2.093665
iteration 1000 / 1500: loss 2.161012
iteration 1100 / 1500: loss 2.113945
iteration 1200 / 1500: loss 2.143529
iteration 1300 / 1500: loss 2.082231
iteration 1400 / 1500: loss 2.146085
iteration 0 / 1500: loss 1239.178520
iteration 100 / 1500: loss 2.087490
iteration 200 / 1500: loss 2.107187
iteration 300 / 1500: loss 2.101655
iteration 400 / 1500: loss 2.126228
iteration 500 / 1500: loss 2.143764
iteration 600 / 1500: loss 2.121496
iteration 700 / 1500: loss 2.081205
iteration 800 / 1500: loss 2.181074
iteration 900 / 1500: loss 2.112314
iteration 1000 / 1500: loss 2.139659
iteration 1100 / 1500: loss 2.107832
iteration 1200 / 1500: loss 2.149560
iteration 1300 / 1500: loss 2.094742
iteration 1400 / 1500: loss 2.103958
lr 8.000000e-08 reg 1.500000e+04 train accuracy: 0.344347 val accuracy: 0.355000
lr 8.000000e-08 reg 1.777778e+04 train accuracy: 0.341837 val accuracy: 0.355000
lr 8.000000e-08 reg 2.055556e+04 train accuracy: 0.333694 val accuracy: 0.344000
lr 8.000000e-08 reg 2.333333e+04 train accuracy: 0.330857 val accuracy: 0.347000
lr 8.000000e-08 reg 2.611111e+04 train accuracy: 0.326306 val accuracy: 0.337000
lr 8.000000e-08 reg 2.888889e+04 train accuracy: 0.325408 val accuracy: 0.352000
lr 8.000000e-08 reg 3.166667e+04 train accuracy: 0.317204 val accuracy: 0.334000
lr 8.000000e-08 reg 3.444444e+04 train accuracy: 0.322102 val accuracy: 0.344000
lr 8.000000e-08 reg 3.722222e+04 train accuracy: 0.315571 val accuracy: 0.333000
lr 8.000000e-08 reg 4.000000e+04 train accuracy: 0.314061 val accuracy: 0.336000
lr 1.822222e-07 reg 1.500000e+04 train accuracy: 0.346082 val accuracy: 0.359000
lr 1.822222e-07 reg 1.777778e+04 train accuracy: 0.339429 val accuracy: 0.351000
lr 1.822222e-07 reg 2.055556e+04 train accuracy: 0.333633 val accuracy: 0.356000
lr 1.822222e-07 reg 2.333333e+04 train accuracy: 0.334102 val accuracy: 0.346000
lr 1.822222e-07 reg 2.611111e+04 train accuracy: 0.316388 val accuracy: 0.329000
lr 1.822222e-07 reg 2.888889e+04 train accuracy: 0.327755 val accuracy: 0.344000
lr 1.822222e-07 reg 3.166667e+04 train accuracy: 0.322714 val accuracy: 0.338000
lr 1.822222e-07 reg 3.444444e+04 train accuracy: 0.324878 val accuracy: 0.344000
lr 1.822222e-07 reg 3.722222e+04 train accuracy: 0.312306 val accuracy: 0.320000
lr 1.822222e-07 reg 4.000000e+04 train accuracy: 0.305714 val accuracy: 0.333000
lr 2.844444e-07 reg 1.500000e+04 train accuracy: 0.347041 val accuracy: 0.359000
lr 2.844444e-07 reg 1.777778e+04 train accuracy: 0.338776 val accuracy: 0.357000
lr 2.844444e-07 reg 2.055556e+04 train accuracy: 0.338796 val accuracy: 0.354000
lr 2.844444e-07 reg 2.333333e+04 train accuracy: 0.334735 val accuracy: 0.339000
lr 2.844444e-07 reg 2.611111e+04 train accuracy: 0.322245 val accuracy: 0.336000
lr 2.844444e-07 reg 2.888889e+04 train accuracy: 0.327102 val accuracy: 0.345000
lr 2.844444e-07 reg 3.166667e+04 train accuracy: 0.318531 val accuracy: 0.329000
lr 2.844444e-07 reg 3.444444e+04 train accuracy: 0.323571 val accuracy: 0.337000
lr 2.844444e-07 reg 3.722222e+04 train accuracy: 0.312959 val accuracy: 0.328000
lr 2.844444e-07 reg 4.000000e+04 train accuracy: 0.316408 val accuracy: 0.328000
lr 3.866667e-07 reg 1.500000e+04 train accuracy: 0.348714 val accuracy: 0.355000
lr 3.866667e-07 reg 1.777778e+04 train accuracy: 0.337000 val accuracy: 0.334000
lr 3.866667e-07 reg 2.055556e+04 train accuracy: 0.331980 val accuracy: 0.351000
lr 3.866667e-07 reg 2.333333e+04 train accuracy: 0.338306 val accuracy: 0.341000
lr 3.866667e-07 reg 2.611111e+04 train accuracy: 0.331776 val accuracy: 0.350000
lr 3.866667e-07 reg 2.888889e+04 train accuracy: 0.314388 val accuracy: 0.329000
```

```
lr 3.866667e-07 reg 3.166667e+04 train accuracy: 0.316959 val accuracy: 0.338000
lr 3.866667e-07 reg 3.444444e+04 train accuracy: 0.310204 val accuracy: 0.332000
lr 3.866667e-07 reg 3.722222e+04 train accuracy: 0.315367 val accuracy: 0.331000
lr 3.866667e-07 reg 4.000000e+04 train accuracy: 0.304061 val accuracy: 0.318000
lr 4.888889e-07 reg 1.500000e+04 train accuracy: 0.349449 val accuracy: 0.356000
lr 4.888889e-07 reg 1.777778e+04 train accuracy: 0.334000 val accuracy: 0.344000
lr 4.888889e-07 reg 2.055556e+04 train accuracy: 0.336837 val accuracy: 0.334000
lr 4.888889e-07 reg 2.333333e+04 train accuracy: 0.322551 val accuracy: 0.337000
lr 4.888889e-07 reg 2.611111e+04 train accuracy: 0.326918 val accuracy: 0.337000
lr 4.888889e-07 reg 2.888889e+04 train accuracy: 0.310245 val accuracy: 0.321000
lr 4.888889e-07 reg 3.166667e+04 train accuracy: 0.316694 val accuracy: 0.314000
lr 4.888889e-07 reg 3.444444e+04 train accuracy: 0.317633 val accuracy: 0.328000
lr 4.888889e-07 reg 3.722222e+04 train accuracy: 0.297673 val accuracy: 0.317000
lr 4.888889e-07 reg 4.000000e+04 train accuracy: 0.308224 val accuracy: 0.327000
lr 5.911111e-07 reg 1.500000e+04 train accuracy: 0.337510 val accuracy: 0.334000
lr 5.911111e-07 reg 1.777778e+04 train accuracy: 0.333980 val accuracy: 0.344000
lr 5.911111e-07 reg 2.055556e+04 train accuracy: 0.324265 val accuracy: 0.344000
lr 5.911111e-07 reg 2.333333e+04 train accuracy: 0.324000 val accuracy: 0.340000
lr 5.911111e-07 reg 2.611111e+04 train accuracy: 0.317612 val accuracy: 0.325000
lr 5.911111e-07 reg 2.888889e+04 train accuracy: 0.314918 val accuracy: 0.324000
lr 5.911111e-07 reg 3.166667e+04 train accuracy: 0.323490 val accuracy: 0.323000
lr 5.911111e-07 reg 3.444444e+04 train accuracy: 0.310714 val accuracy: 0.320000
lr 5.911111e-07 reg 3.722222e+04 train accuracy: 0.318531 val accuracy: 0.323000
lr 5.911111e-07 reg 4.000000e+04 train accuracy: 0.312327 val accuracy: 0.319000
lr 6.933333e-07 reg 1.500000e+04 train accuracy: 0.342327 val accuracy: 0.354000
lr 6.933333e-07 reg 1.777778e+04 train accuracy: 0.335102 val accuracy: 0.353000
lr 6.933333e-07 reg 2.055556e+04 train accuracy: 0.329612 val accuracy: 0.345000
lr 6.933333e-07 reg 2.333333e+04 train accuracy: 0.312714 val accuracy: 0.343000
lr 6.933333e-07 reg 2.611111e+04 train accuracy: 0.312306 val accuracy: 0.310000
lr 6.933333e-07 reg 2.888889e+04 train accuracy: 0.308408 val accuracy: 0.323000
lr 6.933333e-07 reg 3.166667e+04 train accuracy: 0.316735 val accuracy: 0.324000
lr 6.933333e-07 reg 3.444444e+04 train accuracy: 0.311673 val accuracy: 0.321000
lr 6.933333e-07 reg 3.722222e+04 train accuracy: 0.308776 val accuracy: 0.311000
lr 6.933333e-07 reg 4.000000e+04 train accuracy: 0.297388 val accuracy: 0.309000
lr 7.955556e-07 reg 1.500000e+04 train accuracy: 0.348020 val accuracy: 0.357000
lr 7.955556e-07 reg 1.777778e+04 train accuracy: 0.341776 val accuracy: 0.344000
lr 7.955556e-07 reg 2.055556e+04 train accuracy: 0.332816 val accuracy: 0.333000
lr 7.955556e-07 reg 2.333333e+04 train accuracy: 0.329939 val accuracy: 0.346000
lr 7.955556e-07 reg 2.611111e+04 train accuracy: 0.326082 val accuracy: 0.334000
lr 7.955556e-07 reg 2.888889e+04 train accuracy: 0.321102 val accuracy: 0.340000
lr 7.955556e-07 reg 3.166667e+04 train accuracy: 0.309694 val accuracy: 0.320000
lr 7.955556e-07 reg 3.444444e+04 train accuracy: 0.312592 val accuracy: 0.332000
lr 7.955556e-07 reg 3.722222e+04 train accuracy: 0.309449 val accuracy: 0.339000
lr 7.955556e-07 reg 4.000000e+04 train accuracy: 0.302918 val accuracy: 0.322000
lr 8.977778e-07 reg 1.500000e+04 train accuracy: 0.337816 val accuracy: 0.337000
lr 8.977778e-07 reg 1.777778e+04 train accuracy: 0.328776 val accuracy: 0.351000
lr 8.977778e-07 reg 2.055556e+04 train accuracy: 0.335184 val accuracy: 0.344000
lr 8.977778e-07 reg 2.333333e+04 train accuracy: 0.320939 val accuracy: 0.330000
lr 8.977778e-07 reg 2.611111e+04 train accuracy: 0.317939 val accuracy: 0.330000
lr 8.977778e-07 reg 2.888889e+04 train accuracy: 0.309551 val accuracy: 0.328000
lr 8.977778e-07 reg 3.166667e+04 train accuracy: 0.316633 val accuracy: 0.342000
lr 8.977778e-07 reg 3.444444e+04 train accuracy: 0.294653 val accuracy: 0.312000
lr 8.977778e-07 reg 3.722222e+04 train accuracy: 0.315735 val accuracy: 0.333000
lr 8.977778e-07 reg 4.000000e+04 train accuracy: 0.301898 val accuracy: 0.314000
lr 1.000000e-06 reg 1.500000e+04 train accuracy: 0.344265 val accuracy: 0.350000
lr 1.000000e-06 reg 1.777778e+04 train accuracy: 0.328327 val accuracy: 0.340000
lr 1.000000e-06 reg 2.055556e+04 train accuracy: 0.332694 val accuracy: 0.343000
lr 1.000000e-06 reg 2.333333e+04 train accuracy: 0.328449 val accuracy: 0.344000
lr 1.000000e-06 reg 2.611111e+04 train accuracy: 0.313857 val accuracy: 0.330000
lr 1.000000e-06 reg 2.888889e+04 train accuracy: 0.318388 val accuracy: 0.326000
lr 1.000000e-06 reg 3.166667e+04 train accuracy: 0.301327 val accuracy: 0.310000
lr 1.000000e-06 reg 3.444444e+04 train accuracy: 0.304224 val accuracy: 0.331000
lr 1.000000e-06 reg 3.722222e+04 train accuracy: 0.306571 val accuracy: 0.310000
```

```
lr 1.000000e-06 reg 4.000000e+04 train accuracy: 0.310327 val accuracy: 0.315000
best validation accuracy achieved during cross-validation: 0.359000
```

In [8]:

```
# 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: 0.352000
```

Inline Question - *True or False*

It's possible to add a new data point 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: The loss calculated in SVM only considers categories that do not have a score lower than the correct category score Δ, so adding new data points will not change the loss as long as the score of the wrong category is lower than the score Δ of the correct category. But softmax is different. Adding new data points means reducing the score of the correct category, and the loss function of softmax is associated with the relationship between the correct category and the total score, so the loss will change.

In [9]:

```
# 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', 't
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])
```


```
In [ ]:
```

In [1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment1/'
# FOLDERNAME = 'assignment1'
# assert FOLDERNAME is not None, "[!] Enter the foldername."

# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME

# # Install requirements from colab_requirements.txt
# # TODO: Please change your path below to the colab_requirements.txt file
# ! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/colab_requirements
```

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.

In [2]:
```
# A bit of setup

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs6353.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 `cs6353/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.

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

Forward pass: compute scores

Open the file `cs6353/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.

In [4]:
```
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
```

Forward pass: compute loss

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

In [5]:
```
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
```

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:

In [6]:
```
from cs6353.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=Fal
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads
```

```
W1 max relative error: 3.561318e-09
W2 max relative error: 3.440708e-09
b1 max relative error: 2.738421e-09
b2 max relative error: 4.447656e-11
```

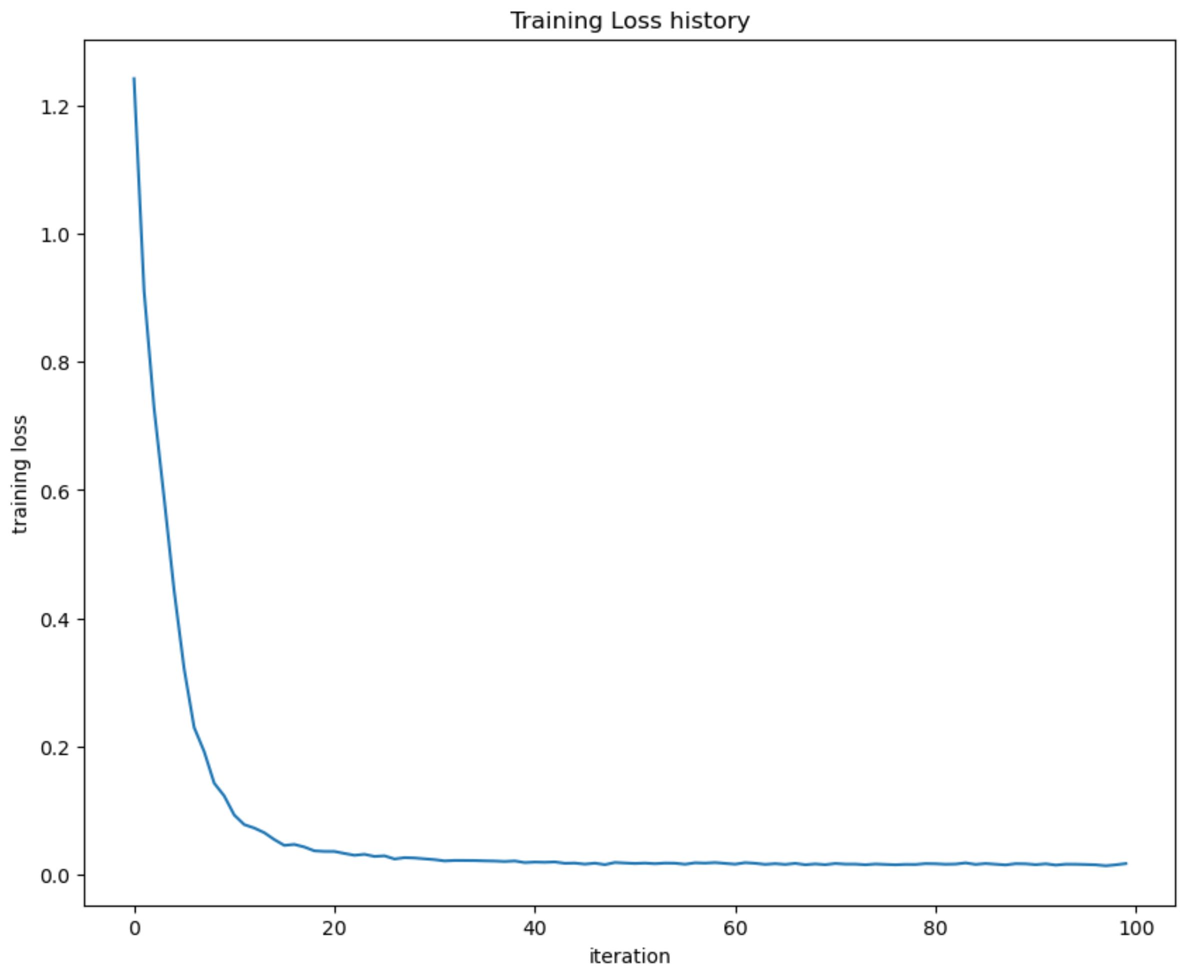
Train the network

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.

In [7]:
```
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:  0.017149607938732048
```



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.

In [8]:
```
from cs6353.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 = 'cs6353/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 memo
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,)
```

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.

In [9]:
```
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
```

Debug the training

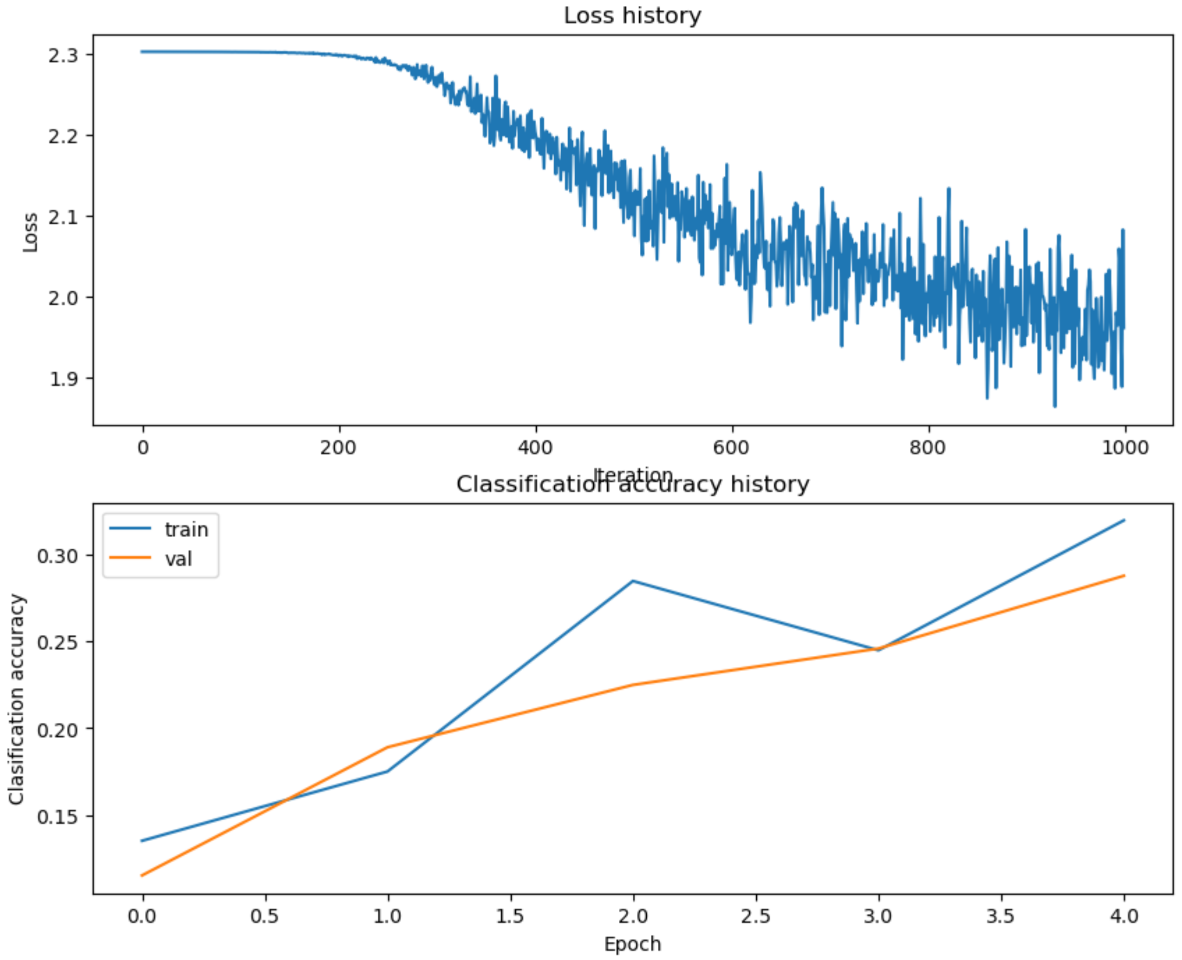
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

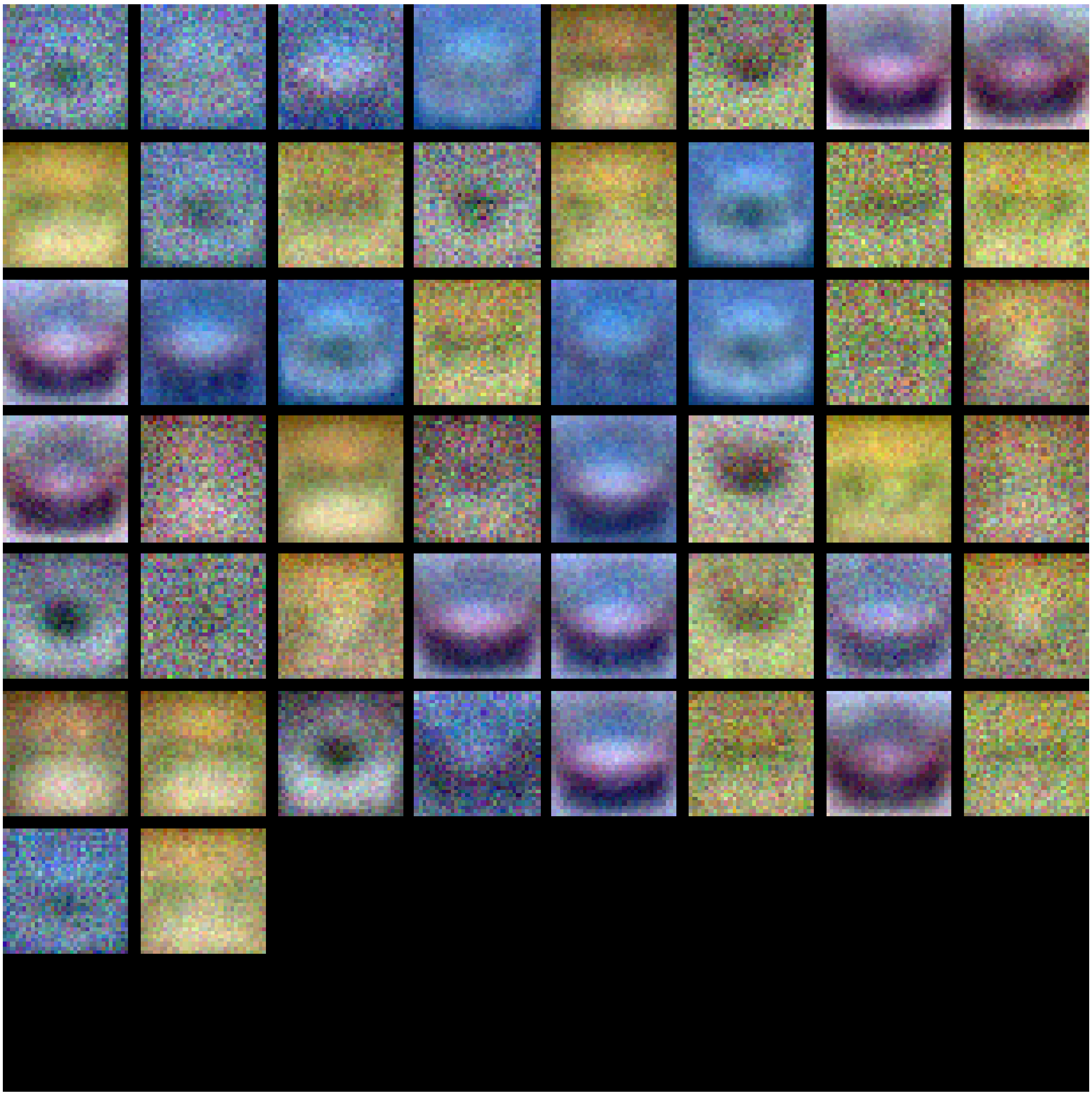
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [10]:
```
# 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()
```



In [11]:
```
from cs6353.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)
```



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, number 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.

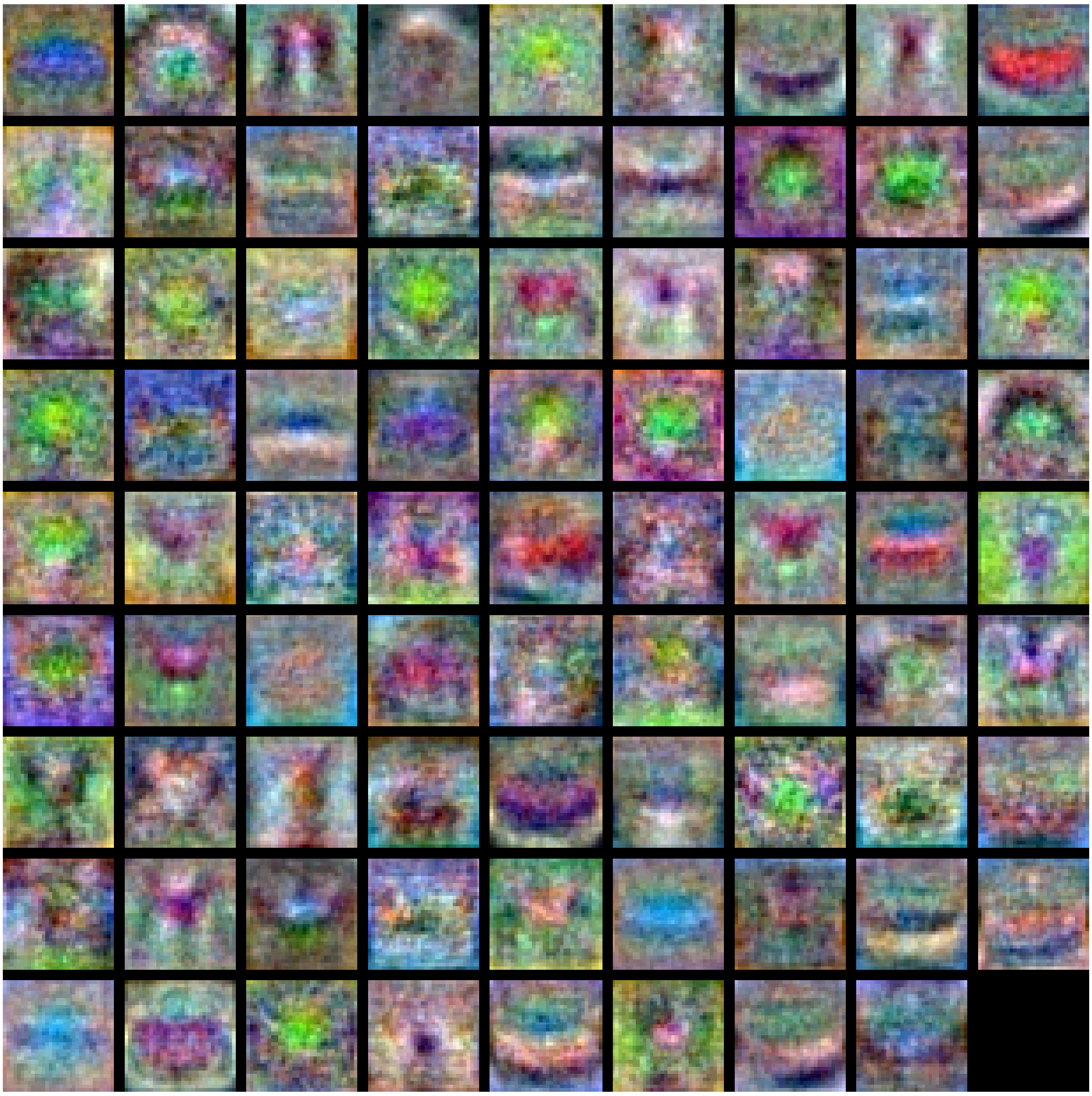
Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [12]:
```
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_acc = 0.0
input_size = 32 * 32 * 3
hidden_size = [70, 80] #
num_classes = 10
learning_rate = [8e-4, 9e-4] #
batch_size = [500, 600] #
num_iters = [1500] #
reg = [0.35, 0.45] #

for lr in learning_rate:
    for size in batch_size:
        for num in num_iters:
            for hi_size in hidden_size:
                for rag in reg:
                    net = TwoLayerNet(input_size, hi_size, num_classes)
                    # Train the network
                    stats = net.train(X_train, y_train, X_val, y_val,
                                num_iters=num, batch_size=size,
                                learning_rate=lr, learning_rate_decay=0.95,
                                reg=rag, verbose=False)
                    # Predict on the validation set
                    val_acc = (net.predict(X_val) == y_val).mean()
                    if val_acc > best_acc:
                        best_acc = val_acc
                        best_net = net
                    print('Validation accuracy: ', val_acc, ' with learning_rate: ',

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

```
Validation accuracy:  0.486  with learning_rate:  0.0008  batch_size:  500  num_iter
s:  1500  hidden_size:  70  reg:  0.35
Validation accuracy:  0.475  with learning_rate:  0.0008  batch_size:  500  num_iter
s:  1500  hidden_size:  70  reg:  0.45
Validation accuracy:  0.478  with learning_rate:  0.0008  batch_size:  500  num_iter
s:  1500  hidden_size:  80  reg:  0.35
Validation accuracy:  0.474  with learning_rate:  0.0008  batch_size:  500  num_iter
s:  1500  hidden_size:  80  reg:  0.45
Validation accuracy:  0.496  with learning_rate:  0.0008  batch_size:  600  num_iter
s:  1500  hidden_size:  70  reg:  0.35
Validation accuracy:  0.492  with learning_rate:  0.0008  batch_size:  600  num_iter
s:  1500  hidden_size:  70  reg:  0.45
Validation accuracy:  0.498  with learning_rate:  0.0008  batch_size:  600  num_iter
s:  1500  hidden_size:  80  reg:  0.35
Validation accuracy:  0.498  with learning_rate:  0.0008  batch_size:  600  num_iter
s:  1500  hidden_size:  80  reg:  0.45
Validation accuracy:  0.5  with learning_rate:  0.0009  batch_size:  500  num_iters:
1500  hidden_size:  70  reg:  0.35
Validation accuracy:  0.484  with learning_rate:  0.0009  batch_size:  500  num_iter
s:  1500  hidden_size:  70  reg:  0.45
Validation accuracy:  0.485  with learning_rate:  0.0009  batch_size:  500  num_iter
s:  1500  hidden_size:  80  reg:  0.35
Validation accuracy:  0.491  with learning_rate:  0.0009  batch_size:  500  num_iter
s:  1500  hidden_size:  80  reg:  0.45
Validation accuracy:  0.487  with learning_rate:  0.0009  batch_size:  600  num_iter
s:  1500  hidden_size:  70  reg:  0.35
Validation accuracy:  0.474  with learning_rate:  0.0009  batch_size:  600  num_iter
s:  1500  hidden_size:  70  reg:  0.45
Validation accuracy:  0.511  with learning_rate:  0.0009  batch_size:  600  num_iter
s:  1500  hidden_size:  80  reg:  0.35
Validation accuracy:  0.489  with learning_rate:  0.0009  batch_size:  600  num_iter
s:  1500  hidden_size:  80  reg:  0.45
```

In [13]:

```
# visualize the weights of the best network
show_net_weights(best_net)
```



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%.

In [14]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.513
```

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 and 3

Your explanation: Test accuracy is lower than training accuracy, which means overfitting occurs. Overfitting can be improved by using a larger training set and adding regularization/increasing the regularization strength. But adding more hidden points will aggravate overfitting and is generally used to solve underfitting problems.

In [ ]:

```

```

In [1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment1/'
# FOLDERNAME = 'assignment1'
# assert FOLDERNAME is not None, "[!] Enter the foldername."

# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME

# # Install requirements from colab_requirements.txt
# # TODO: Please change your path below to the colab_requirements.txt file
# ! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/colab_requirements
```

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.

In [2]:
```
from __future__ import print_function
import random
import numpy as np
from cs6353.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
```

Load data

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

In [3]:
```
from cs6353.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(cifar10_dir='cs6353/datasets/cifar-10-batches-py', num_training
    # Load the raw CIFAR-10 data
    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 memo
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

################################################################################
# TODO: Change the path of the CIFAR-10 data directory correctly to           #
# the correct location                                                         #
# Default path is set to cs6353/datasets/cifar-10-batches-py'                #
################################################################################
cifar10_dir='cs6353/datasets/cifar-10-batches-py'
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir)
```

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.

In [4]:
```
from cs6353.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_bin
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
```

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.

In [5]:

```python
# Use the validation set to tune the learning rate and regularization strength

from cs6353.classifiers.linear_classifier import LinearSVM

learning_rates = np.linspace(1e-7, 9e-7, 8)
regularization_strengths = np.linspace(3e4, 4e4, 3)
```

```
# 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 classifier 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 learn in learning_rates:
    for regular in regularization_strengths:
        new_svm = LinearSVM()
        new_svm.train(X_train_feats, y_train, learn, regular, num_iters=1500, verbos
        y_train_pred = new_svm.predict(X_train_feats)
        y_val_pred = new_svm.predict(X_val_feats)
        train_acc = np.mean(y_train == y_train_pred)
        val_acc = np.mean(y_val == y_val_pred)
        if best_val < val_acc:
            best_val = val_acc
            best_svm = new_svm
        results[(learn, regular)] = (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)
```

```
iteration 0 / 1500: loss 56.914025
iteration 100 / 1500: loss 23.376865
iteration 200 / 1500: loss 13.314719
iteration 300 / 1500: loss 10.295868
iteration 400 / 1500: loss 9.388885
iteration 500 / 1500: loss 9.115754
iteration 600 / 1500: loss 9.034414
iteration 700 / 1500: loss 9.009948
iteration 800 / 1500: loss 9.002526
iteration 900 / 1500: loss 9.000238
iteration 1000 / 1500: loss 8.999638
iteration 1100 / 1500: loss 8.999529
iteration 1200 / 1500: loss 8.999406
iteration 1300 / 1500: loss 8.999391
iteration 1400 / 1500: loss 8.999427
iteration 0 / 1500: loss 63.759720
iteration 100 / 1500: loss 22.434256
iteration 200 / 1500: loss 12.296035
iteration 300 / 1500: loss 9.809327
iteration 400 / 1500: loss 9.199019
iteration 500 / 1500: loss 9.047974
iteration 600 / 1500: loss 9.011490
iteration 700 / 1500: loss 9.002567
iteration 800 / 1500: loss 9.000141
iteration 900 / 1500: loss 8.999533
iteration 1000 / 1500: loss 8.999575
iteration 1100 / 1500: loss 8.999509
iteration 1200 / 1500: loss 8.999491
iteration 1300 / 1500: loss 8.999577
iteration 1400 / 1500: loss 8.999483
iteration 0 / 1500: loss 68.556404
iteration 100 / 1500: loss 20.944379
iteration 200 / 1500: loss 11.397874
iteration 300 / 1500: loss 9.480094
iteration 400 / 1500: loss 9.095764
iteration 500 / 1500: loss 9.018983
iteration 600 / 1500: loss 9.003338
iteration 700 / 1500: loss 9.000258
iteration 800 / 1500: loss 8.999759
iteration 900 / 1500: loss 8.999513
iteration 1000 / 1500: loss 8.999551
iteration 1100 / 1500: loss 8.999618
iteration 1200 / 1500: loss 8.999518
iteration 1300 / 1500: loss 8.999561
iteration 1400 / 1500: loss 8.999589
iteration 0 / 1500: loss 56.002733
iteration 100 / 1500: loss 12.531411
iteration 200 / 1500: loss 9.265097
iteration 300 / 1500: loss 9.019208
iteration 400 / 1500: loss 9.000988
iteration 500 / 1500: loss 8.999433
iteration 600 / 1500: loss 8.999293
iteration 700 / 1500: loss 8.999509
iteration 800 / 1500: loss 8.999354
iteration 900 / 1500: loss 8.999383
iteration 1000 / 1500: loss 8.999461
iteration 1100 / 1500: loss 8.999431
iteration 1200 / 1500: loss 8.999212
iteration 1300 / 1500: loss 8.999522
iteration 1400 / 1500: loss 8.999411
iteration 0 / 1500: loss 62.391956
iteration 100 / 1500: loss 11.595681
iteration 200 / 1500: loss 9.126063
iteration 300 / 1500: loss 9.005546
```

```
iteration 400 / 1500: loss 8.999848
iteration 500 / 1500: loss 8.999616
iteration 600 / 1500: loss 8.999556
iteration 700 / 1500: loss 8.999409
iteration 800 / 1500: loss 8.999583
iteration 900 / 1500: loss 8.999473
iteration 1000 / 1500: loss 8.999486
iteration 1100 / 1500: loss 8.999441
iteration 1200 / 1500: loss 8.999494
iteration 1300 / 1500: loss 8.999428
iteration 1400 / 1500: loss 8.999586
iteration 0 / 1500: loss 71.976473
iteration 100 / 1500: loss 10.980199
iteration 200 / 1500: loss 9.061783
iteration 300 / 1500: loss 9.001445
iteration 400 / 1500: loss 8.999576
iteration 500 / 1500: loss 8.999553
iteration 600 / 1500: loss 8.999587
iteration 700 / 1500: loss 8.999462
iteration 800 / 1500: loss 8.999526
iteration 900 / 1500: loss 8.999550
iteration 1000 / 1500: loss 8.999547
iteration 1100 / 1500: loss 8.999518
iteration 1200 / 1500: loss 8.999522
iteration 1300 / 1500: loss 8.999625
iteration 1400 / 1500: loss 8.999533
iteration 0 / 1500: loss 55.961410
iteration 100 / 1500: loss 9.874959
iteration 200 / 1500: loss 9.015785
iteration 300 / 1500: loss 8.999565
iteration 400 / 1500: loss 8.999404
iteration 500 / 1500: loss 8.999414
iteration 600 / 1500: loss 8.999448
iteration 700 / 1500: loss 8.999350
iteration 800 / 1500: loss 8.999365
iteration 900 / 1500: loss 8.999477
iteration 1000 / 1500: loss 8.999341
iteration 1100 / 1500: loss 8.999393
iteration 1200 / 1500: loss 8.999409
iteration 1300 / 1500: loss 8.999374
iteration 1400 / 1500: loss 8.999427
iteration 0 / 1500: loss 63.402026
iteration 100 / 1500: loss 9.517690
iteration 200 / 1500: loss 9.004477
iteration 300 / 1500: loss 8.999440
iteration 400 / 1500: loss 8.999486
iteration 500 / 1500: loss 8.999488
iteration 600 / 1500: loss 8.999590
iteration 700 / 1500: loss 8.999518
iteration 800 / 1500: loss 8.999537
iteration 900 / 1500: loss 8.999542
iteration 1000 / 1500: loss 8.999454
iteration 1100 / 1500: loss 8.999444
iteration 1200 / 1500: loss 8.999517
iteration 1300 / 1500: loss 8.999509
iteration 1400 / 1500: loss 8.999485
iteration 0 / 1500: loss 68.759685
iteration 100 / 1500: loss 9.290477
iteration 200 / 1500: loss 9.000905
iteration 300 / 1500: loss 8.999613
iteration 400 / 1500: loss 8.999620
iteration 500 / 1500: loss 8.999523
iteration 600 / 1500: loss 8.999615
iteration 700 / 1500: loss 8.999608
```

```
iteration 800 / 1500: loss 8.999589
iteration 900 / 1500: loss 8.999519
iteration 1000 / 1500: loss 8.999545
iteration 1100 / 1500: loss 8.999439
iteration 1200 / 1500: loss 8.999545
iteration 1300 / 1500: loss 8.999581
iteration 1400 / 1500: loss 8.999483
iteration 0 / 1500: loss 54.774862
iteration 100 / 1500: loss 9.208655
iteration 200 / 1500: loss 9.000484
iteration 300 / 1500: loss 8.999350
iteration 400 / 1500: loss 8.999471
iteration 500 / 1500: loss 8.999379
iteration 600 / 1500: loss 8.999398
iteration 700 / 1500: loss 8.999406
iteration 800 / 1500: loss 8.999335
iteration 900 / 1500: loss 8.999524
iteration 1000 / 1500: loss 8.999394
iteration 1100 / 1500: loss 8.999409
iteration 1200 / 1500: loss 8.999424
iteration 1300 / 1500: loss 8.999224
iteration 1400 / 1500: loss 8.999402
iteration 0 / 1500: loss 64.206213
iteration 100 / 1500: loss 9.101052
iteration 200 / 1500: loss 8.999694
iteration 300 / 1500: loss 8.999668
iteration 400 / 1500: loss 8.999562
iteration 500 / 1500: loss 8.999470
iteration 600 / 1500: loss 8.999453
iteration 700 / 1500: loss 8.999432
iteration 800 / 1500: loss 8.999517
iteration 900 / 1500: loss 8.999387
iteration 1000 / 1500: loss 8.999486
iteration 1100 / 1500: loss 8.999628
iteration 1200 / 1500: loss 8.999540
iteration 1300 / 1500: loss 8.999529
iteration 1400 / 1500: loss 8.999452
iteration 0 / 1500: loss 69.245668
iteration 100 / 1500: loss 9.043942
iteration 200 / 1500: loss 8.999711
iteration 300 / 1500: loss 8.999455
iteration 400 / 1500: loss 8.999554
iteration 500 / 1500: loss 8.999534
iteration 600 / 1500: loss 8.999592
iteration 700 / 1500: loss 8.999609
iteration 800 / 1500: loss 8.999592
iteration 900 / 1500: loss 8.999511
iteration 1000 / 1500: loss 8.999525
iteration 1100 / 1500: loss 8.999578
iteration 1200 / 1500: loss 8.999616
iteration 1300 / 1500: loss 8.999524
iteration 1400 / 1500: loss 8.999620
iteration 0 / 1500: loss 53.697677
iteration 100 / 1500: loss 9.049369
iteration 200 / 1500: loss 8.999372
iteration 300 / 1500: loss 8.999425
iteration 400 / 1500: loss 8.999557
iteration 500 / 1500: loss 8.999459
iteration 600 / 1500: loss 8.999502
iteration 700 / 1500: loss 8.999315
iteration 800 / 1500: loss 8.999441
iteration 900 / 1500: loss 8.999513
iteration 1000 / 1500: loss 8.999441
iteration 1100 / 1500: loss 8.999259
```

```
iteration 1200 / 1500: loss 8.999332
iteration 1300 / 1500: loss 8.999483
iteration 1400 / 1500: loss 8.999308
iteration 0 / 1500: loss 64.241797
iteration 100 / 1500: loss 9.018860
iteration 200 / 1500: loss 8.999637
iteration 300 / 1500: loss 8.999546
iteration 400 / 1500: loss 8.999401
iteration 500 / 1500: loss 8.999525
iteration 600 / 1500: loss 8.999488
iteration 700 / 1500: loss 8.999534
iteration 800 / 1500: loss 8.999512
iteration 900 / 1500: loss 8.999584
iteration 1000 / 1500: loss 8.999560
iteration 1100 / 1500: loss 8.999496
iteration 1200 / 1500: loss 8.999526
iteration 1300 / 1500: loss 8.999395
iteration 1400 / 1500: loss 8.999567
iteration 0 / 1500: loss 72.012904
iteration 100 / 1500: loss 9.006531
iteration 200 / 1500: loss 8.999613
iteration 300 / 1500: loss 8.999555
iteration 400 / 1500: loss 8.999566
iteration 500 / 1500: loss 8.999501
iteration 600 / 1500: loss 8.999546
iteration 700 / 1500: loss 8.999562
iteration 800 / 1500: loss 8.999641
iteration 900 / 1500: loss 8.999626
iteration 1000 / 1500: loss 8.999566
iteration 1100 / 1500: loss 8.999612
iteration 1200 / 1500: loss 8.999638
iteration 1300 / 1500: loss 8.999522
iteration 1400 / 1500: loss 8.999536
iteration 0 / 1500: loss 57.129250
iteration 100 / 1500: loss 9.012166
iteration 200 / 1500: loss 8.999473
iteration 300 / 1500: loss 8.999519
iteration 400 / 1500: loss 8.999362
iteration 500 / 1500: loss 8.999474
iteration 600 / 1500: loss 8.999472
iteration 700 / 1500: loss 8.999485
iteration 800 / 1500: loss 8.999387
iteration 900 / 1500: loss 8.999303
iteration 1000 / 1500: loss 8.999462
iteration 1100 / 1500: loss 8.999503
iteration 1200 / 1500: loss 8.999416
iteration 1300 / 1500: loss 8.999357
iteration 1400 / 1500: loss 8.999459
iteration 0 / 1500: loss 61.047953
iteration 100 / 1500: loss 9.002960
iteration 200 / 1500: loss 8.999483
iteration 300 / 1500: loss 8.999526
iteration 400 / 1500: loss 8.999423
iteration 500 / 1500: loss 8.999640
iteration 600 / 1500: loss 8.999425
iteration 700 / 1500: loss 8.999407
iteration 800 / 1500: loss 8.999497
iteration 900 / 1500: loss 8.999597
iteration 1000 / 1500: loss 8.999437
iteration 1100 / 1500: loss 8.999460
iteration 1200 / 1500: loss 8.999453
iteration 1300 / 1500: loss 8.999508
iteration 1400 / 1500: loss 8.999491
iteration 0 / 1500: loss 70.955597
```

```
iteration 100 / 1500: loss 9.000539
iteration 200 / 1500: loss 8.999590
iteration 300 / 1500: loss 8.999539
iteration 400 / 1500: loss 8.999592
iteration 500 / 1500: loss 8.999517
iteration 600 / 1500: loss 8.999630
iteration 700 / 1500: loss 8.999552
iteration 800 / 1500: loss 8.999549
iteration 900 / 1500: loss 8.999566
iteration 1000 / 1500: loss 8.999536
iteration 1100 / 1500: loss 8.999584
iteration 1200 / 1500: loss 8.999528
iteration 1300 / 1500: loss 8.999610
iteration 1400 / 1500: loss 8.999576
iteration 0 / 1500: loss 54.594498
iteration 100 / 1500: loss 9.002398
iteration 200 / 1500: loss 8.999382
iteration 300 / 1500: loss 8.999484
iteration 400 / 1500: loss 8.999318
iteration 500 / 1500: loss 8.999343
iteration 600 / 1500: loss 8.999420
iteration 700 / 1500: loss 8.999335
iteration 800 / 1500: loss 8.999334
iteration 900 / 1500: loss 8.999503
iteration 1000 / 1500: loss 8.999467
iteration 1100 / 1500: loss 8.999524
iteration 1200 / 1500: loss 8.999350
iteration 1300 / 1500: loss 8.999385
iteration 1400 / 1500: loss 8.999378
iteration 0 / 1500: loss 64.494798
iteration 100 / 1500: loss 9.000060
iteration 200 / 1500: loss 8.999565
iteration 300 / 1500: loss 8.999589
iteration 400 / 1500: loss 8.999465
iteration 500 / 1500: loss 8.999406
iteration 600 / 1500: loss 8.999439
iteration 700 / 1500: loss 8.999556
iteration 800 / 1500: loss 8.999445
iteration 900 / 1500: loss 8.999560
iteration 1000 / 1500: loss 8.999442
iteration 1100 / 1500: loss 8.999578
iteration 1200 / 1500: loss 8.999448
iteration 1300 / 1500: loss 8.999498
iteration 1400 / 1500: loss 8.999524
iteration 0 / 1500: loss 71.123624
iteration 100 / 1500: loss 8.999722
iteration 200 / 1500: loss 8.999564
iteration 300 / 1500: loss 8.999615
iteration 400 / 1500: loss 8.999658
iteration 500 / 1500: loss 8.999547
iteration 600 / 1500: loss 8.999624
iteration 700 / 1500: loss 8.999683
iteration 800 / 1500: loss 8.999611
iteration 900 / 1500: loss 8.999570
iteration 1000 / 1500: loss 8.999418
iteration 1100 / 1500: loss 8.999541
iteration 1200 / 1500: loss 8.999554
iteration 1300 / 1500: loss 8.999556
iteration 1400 / 1500: loss 8.999594
iteration 0 / 1500: loss 54.958600
iteration 100 / 1500: loss 9.000150
iteration 200 / 1500: loss 8.999425
iteration 300 / 1500: loss 8.999396
iteration 400 / 1500: loss 8.999561
```

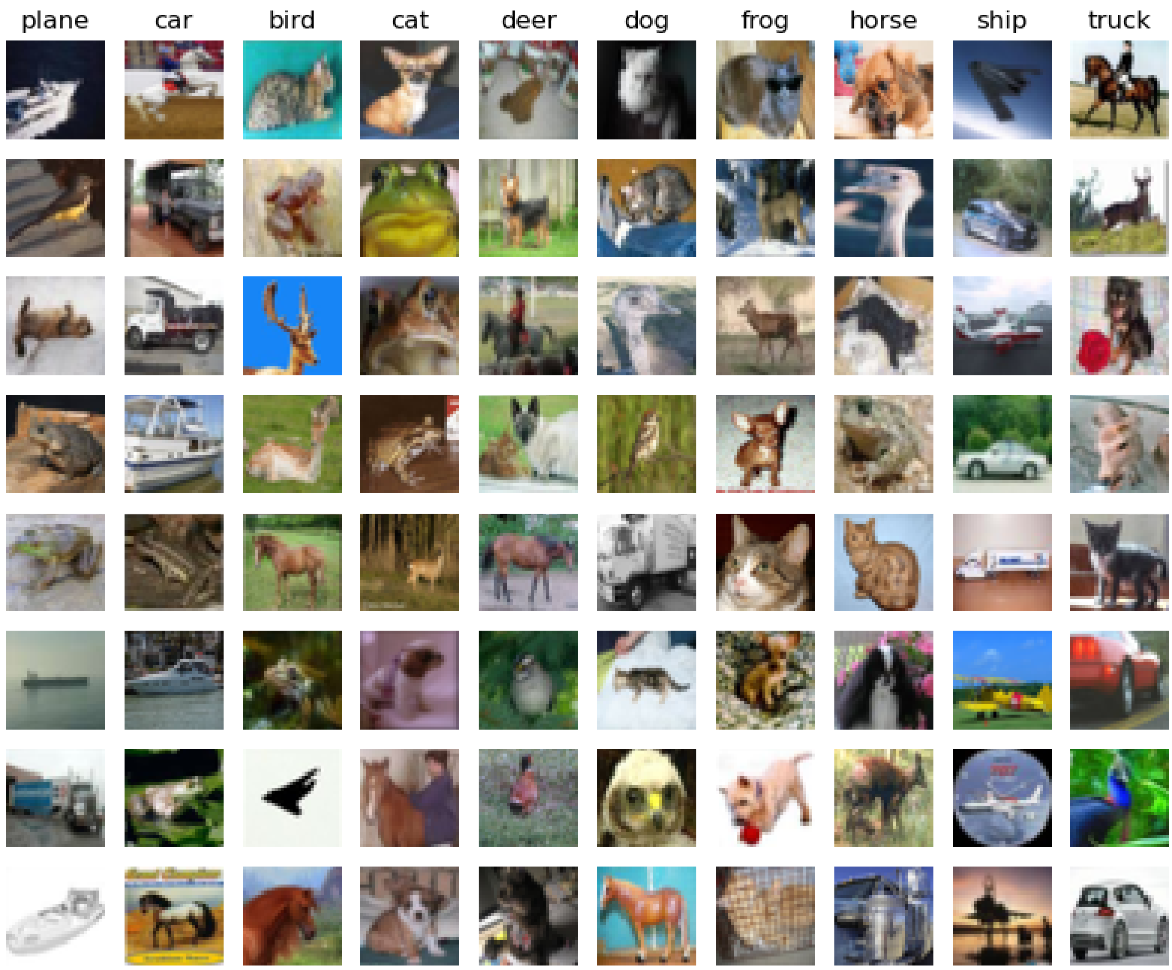
```
iteration 500 / 1500: loss 8.999476
iteration 600 / 1500: loss 8.999472
iteration 700 / 1500: loss 8.999435
iteration 800 / 1500: loss 8.999371
iteration 900 / 1500: loss 8.999520
iteration 1000 / 1500: loss 8.999404
iteration 1100 / 1500: loss 8.999468
iteration 1200 / 1500: loss 8.999398
iteration 1300 / 1500: loss 8.999412
iteration 1400 / 1500: loss 8.999384
iteration 0 / 1500: loss 63.604936
iteration 100 / 1500: loss 8.999491
iteration 200 / 1500: loss 8.999551
iteration 300 / 1500: loss 8.999578
iteration 400 / 1500: loss 8.999493
iteration 500 / 1500: loss 8.999560
iteration 600 / 1500: loss 8.999473
iteration 700 / 1500: loss 8.999532
iteration 800 / 1500: loss 8.999451
iteration 900 / 1500: loss 8.999519
iteration 1000 / 1500: loss 8.999597
iteration 1100 / 1500: loss 8.999368
iteration 1200 / 1500: loss 8.999448
iteration 1300 / 1500: loss 8.999491
iteration 1400 / 1500: loss 8.999511
iteration 0 / 1500: loss 72.609216
iteration 100 / 1500: loss 8.999677
iteration 200 / 1500: loss 8.999629
iteration 300 / 1500: loss 8.999520
iteration 400 / 1500: loss 8.999491
iteration 500 / 1500: loss 8.999587
iteration 600 / 1500: loss 8.999653
iteration 700 / 1500: loss 8.999685
iteration 800 / 1500: loss 8.999620
iteration 900 / 1500: loss 8.999587
iteration 1000 / 1500: loss 8.999539
iteration 1100 / 1500: loss 8.999627
iteration 1200 / 1500: loss 8.999514
iteration 1300 / 1500: loss 8.999609
iteration 1400 / 1500: loss 8.999529
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.415918 val accuracy: 0.425000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.416816 val accuracy: 0.417000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.415020 val accuracy: 0.415000
lr 2.142857e-07 reg 3.000000e+04 train accuracy: 0.417469 val accuracy: 0.425000
lr 2.142857e-07 reg 3.500000e+04 train accuracy: 0.413429 val accuracy: 0.403000
lr 2.142857e-07 reg 4.000000e+04 train accuracy: 0.415735 val accuracy: 0.419000
lr 3.285714e-07 reg 3.000000e+04 train accuracy: 0.414918 val accuracy: 0.408000
lr 3.285714e-07 reg 3.500000e+04 train accuracy: 0.411082 val accuracy: 0.413000
lr 3.285714e-07 reg 4.000000e+04 train accuracy: 0.409571 val accuracy: 0.408000
lr 4.428571e-07 reg 3.000000e+04 train accuracy: 0.417347 val accuracy: 0.412000
lr 4.428571e-07 reg 3.500000e+04 train accuracy: 0.413878 val accuracy: 0.420000
lr 4.428571e-07 reg 4.000000e+04 train accuracy: 0.408633 val accuracy: 0.401000
lr 5.571429e-07 reg 3.000000e+04 train accuracy: 0.412653 val accuracy: 0.413000
lr 5.571429e-07 reg 3.500000e+04 train accuracy: 0.414020 val accuracy: 0.411000
lr 5.571429e-07 reg 4.000000e+04 train accuracy: 0.415898 val accuracy: 0.421000
lr 6.714286e-07 reg 3.000000e+04 train accuracy: 0.412551 val accuracy: 0.415000
lr 6.714286e-07 reg 3.500000e+04 train accuracy: 0.413204 val accuracy: 0.410000
lr 6.714286e-07 reg 4.000000e+04 train accuracy: 0.402286 val accuracy: 0.395000
lr 7.857143e-07 reg 3.000000e+04 train accuracy: 0.408939 val accuracy: 0.412000
lr 7.857143e-07 reg 3.500000e+04 train accuracy: 0.412857 val accuracy: 0.406000
lr 7.857143e-07 reg 4.000000e+04 train accuracy: 0.410510 val accuracy: 0.409000
lr 9.000000e-07 reg 3.000000e+04 train accuracy: 0.410000 val accuracy: 0.413000
lr 9.000000e-07 reg 3.500000e+04 train accuracy: 0.400633 val accuracy: 0.391000
```

```
lr 9.000000e-07 reg 4.000000e+04 train accuracy: 0.409388 val accuracy: 0.420000
best validation accuracy achieved during cross-validation: 0.425000
```

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

```
0.424
```

In [7]:
```
# 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', 't
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()
```



Inline question 1:

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

There are some very different pictures that are classified into the wrong categories. For example, under the category of birds, horse heads, cats, deer heads, and frogs appear, but they are all very blurry and noisy. For example, there are only photos of the head instead of the whole body, only the side profile and the strange pose of the animal while sleeping instead of the normal standing frontal image of the animal. In addition, very blurry images are also more likely to be misclassified.

Neural Network on image features

Earlier in this assignment 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.

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

In [9]:
```
from cs6353.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]

best_net = None

################################################################################
# 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.                                              #
################################################################################

best_acc = 0.0
hidden_size = [50, 80, 110, 140]
num_classes = 10
learning_rate = [0.1, 0.2, 0.3]
batch_size = [200, 400, 600]
num_iters = [1500]
reg = [1e-05, 1e-06, 1e-07]

for lr in learning_rate:
    for size in batch_size:
        for num in num_iters:
            for hi_size in hidden_size:
                for rag in reg:
                    net = TwoLayerNet(input_dim, hi_size, num_classes)
```

```
                    # Train the network
                    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                                num_iters=num, batch_size=size,
                                learning_rate=lr, learning_rate_decay=0.95,
                                reg=rag, verbose=False)
                    # Predict on the validation set
                    val_acc = (net.predict(X_val_feats) == y_val).mean()
                    if val_acc > best_acc:
                        best_acc = val_acc
                        best_net = net
                    print('Validation accuracy: ', val_acc, ' with learning_rate: ',

                    # # 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()

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

```
Validation accuracy:  0.517  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  50  reg:  1e-05
Validation accuracy:  0.513  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  50  reg:  1e-06
Validation accuracy:  0.531  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  50  reg:  1e-07
Validation accuracy:  0.516  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  80  reg:  1e-05
Validation accuracy:  0.517  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.524  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.525  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.523  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  110  reg:  1e-06
Validation accuracy:  0.521  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  110  reg:  1e-07
Validation accuracy:  0.526  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  140  reg:  1e-05
Validation accuracy:  0.528  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  140  reg:  1e-06
Validation accuracy:  0.522  with learning_rate:  0.1  batch_size:  200  num_iters:
1500  hidden_size:  140  reg:  1e-07
Validation accuracy:  0.519  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  50  reg:  1e-05
Validation accuracy:  0.524  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  50  reg:  1e-06
Validation accuracy:  0.513  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  50  reg:  1e-07
Validation accuracy:  0.515  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  80  reg:  1e-05
Validation accuracy:  0.522  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.527  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.522  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.522  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  110  reg:  1e-06
Validation accuracy:  0.527  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  110  reg:  1e-07
Validation accuracy:  0.517  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  140  reg:  1e-05
Validation accuracy:  0.535  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  140  reg:  1e-06
Validation accuracy:  0.518  with learning_rate:  0.1  batch_size:  400  num_iters:
1500  hidden_size:  140  reg:  1e-07
Validation accuracy:  0.525  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  50  reg:  1e-05
Validation accuracy:  0.518  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  50  reg:  1e-06
Validation accuracy:  0.536  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  50  reg:  1e-07
Validation accuracy:  0.529  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-05
Validation accuracy:  0.534  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.53  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.525  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.539  with learning_rate:  0.1  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-06
```

Validation accuracy: 0.531 with learning_rate: 0.1 batch_size: 600 num_iters: 1500 hidden_size: 110 reg: 1e-07
Validation accuracy: 0.531 with learning_rate: 0.1 batch_size: 600 num_iters: 1500 hidden_size: 140 reg: 1e-05
Validation accuracy: 0.528 with learning_rate: 0.1 batch_size: 600 num_iters: 1500 hidden_size: 140 reg: 1e-06
Validation accuracy: 0.54 with learning_rate: 0.1 batch_size: 600 num_iters: 1500 hidden_size: 140 reg: 1e-07
Validation accuracy: 0.561 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 50 reg: 1e-05
Validation accuracy: 0.541 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 50 reg: 1e-06
Validation accuracy: 0.543 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 50 reg: 1e-07
Validation accuracy: 0.56 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 80 reg: 1e-05
Validation accuracy: 0.537 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 80 reg: 1e-06
Validation accuracy: 0.548 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 80 reg: 1e-07
Validation accuracy: 0.541 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 110 reg: 1e-05
Validation accuracy: 0.555 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 110 reg: 1e-06
Validation accuracy: 0.555 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 110 reg: 1e-07
Validation accuracy: 0.557 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 140 reg: 1e-05
Validation accuracy: 0.558 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 140 reg: 1e-06
Validation accuracy: 0.566 with learning_rate: 0.2 batch_size: 200 num_iters: 1500 hidden_size: 140 reg: 1e-07
Validation accuracy: 0.556 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 50 reg: 1e-05
Validation accuracy: 0.566 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 50 reg: 1e-06
Validation accuracy: 0.546 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 50 reg: 1e-07
Validation accuracy: 0.553 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 80 reg: 1e-05
Validation accuracy: 0.55 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 80 reg: 1e-06
Validation accuracy: 0.552 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 80 reg: 1e-07
Validation accuracy: 0.558 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 110 reg: 1e-05
Validation accuracy: 0.561 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 110 reg: 1e-06
Validation accuracy: 0.565 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 110 reg: 1e-07
Validation accuracy: 0.564 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 140 reg: 1e-05
Validation accuracy: 0.568 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 140 reg: 1e-06
Validation accuracy: 0.56 with learning_rate: 0.2 batch_size: 400 num_iters: 1500 hidden_size: 140 reg: 1e-07
Validation accuracy: 0.562 with learning_rate: 0.2 batch_size: 600 num_iters: 1500 hidden_size: 50 reg: 1e-05
Validation accuracy: 0.55 with learning_rate: 0.2 batch_size: 600 num_iters: 1500 hidden_size: 50 reg: 1e-06
Validation accuracy: 0.561 with learning_rate: 0.2 batch_size: 600 num_iters: 1500 hidden_size: 50 reg: 1e-07
Validation accuracy: 0.568 with learning_rate: 0.2 batch_size: 600 num_iters: 1500 hidden_size: 80 reg: 1e-05

```
Validation accuracy:  0.555  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.56  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.58  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.56  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-06
Validation accuracy:  0.571  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-07
Validation accuracy:  0.56  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  140  reg:  1e-05
Validation accuracy:  0.564  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  140  reg:  1e-06
Validation accuracy:  0.563  with learning_rate:  0.2  batch_size:  600  num_iters:
1500  hidden_size:  140  reg:  1e-07
Validation accuracy:  0.535  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  50  reg:  1e-05
Validation accuracy:  0.539  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  50  reg:  1e-06
Validation accuracy:  0.546  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  50  reg:  1e-07
Validation accuracy:  0.569  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  80  reg:  1e-05
Validation accuracy:  0.555  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.563  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.566  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.569  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  110  reg:  1e-06
Validation accuracy:  0.573  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  110  reg:  1e-07
Validation accuracy:  0.568  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  140  reg:  1e-05
Validation accuracy:  0.573  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  140  reg:  1e-06
Validation accuracy:  0.573  with learning_rate:  0.3  batch_size:  200  num_iters:
1500  hidden_size:  140  reg:  1e-07
Validation accuracy:  0.562  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  50  reg:  1e-05
Validation accuracy:  0.562  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  50  reg:  1e-06
Validation accuracy:  0.538  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  50  reg:  1e-07
Validation accuracy:  0.573  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  80  reg:  1e-05
Validation accuracy:  0.58  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.565  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.569  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.569  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  110  reg:  1e-06
Validation accuracy:  0.557  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  110  reg:  1e-07
Validation accuracy:  0.581  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  140  reg:  1e-05
Validation accuracy:  0.586  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  140  reg:  1e-06
Validation accuracy:  0.566  with learning_rate:  0.3  batch_size:  400  num_iters:
1500  hidden_size:  140  reg:  1e-07
```

```
Validation accuracy:  0.577  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  50  reg:  1e-05
Validation accuracy:  0.561  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  50  reg:  1e-06
Validation accuracy:  0.556  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  50  reg:  1e-07
Validation accuracy:  0.579  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-05
Validation accuracy:  0.569  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-06
Validation accuracy:  0.593  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  80  reg:  1e-07
Validation accuracy:  0.593  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-05
Validation accuracy:  0.591  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-06
Validation accuracy:  0.573  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  110  reg:  1e-07
Validation accuracy:  0.589  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  140  reg:  1e-05
Validation accuracy:  0.596  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  140  reg:  1e-06
Validation accuracy:  0.582  with learning_rate:  0.3  batch_size:  600  num_iters:
1500  hidden_size:  140  reg:  1e-07
```

In [10]:

```python
# 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)
```

```
0.57
```

In [ ]:

```python

```