knn

September 30, 2023

In [ ]:
```
# 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. 'cs231n/assignments/assignment1/'
FOLDERNAME = None
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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

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

In [12]:
```
# Run some setup code for this notebook.
```

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

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

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

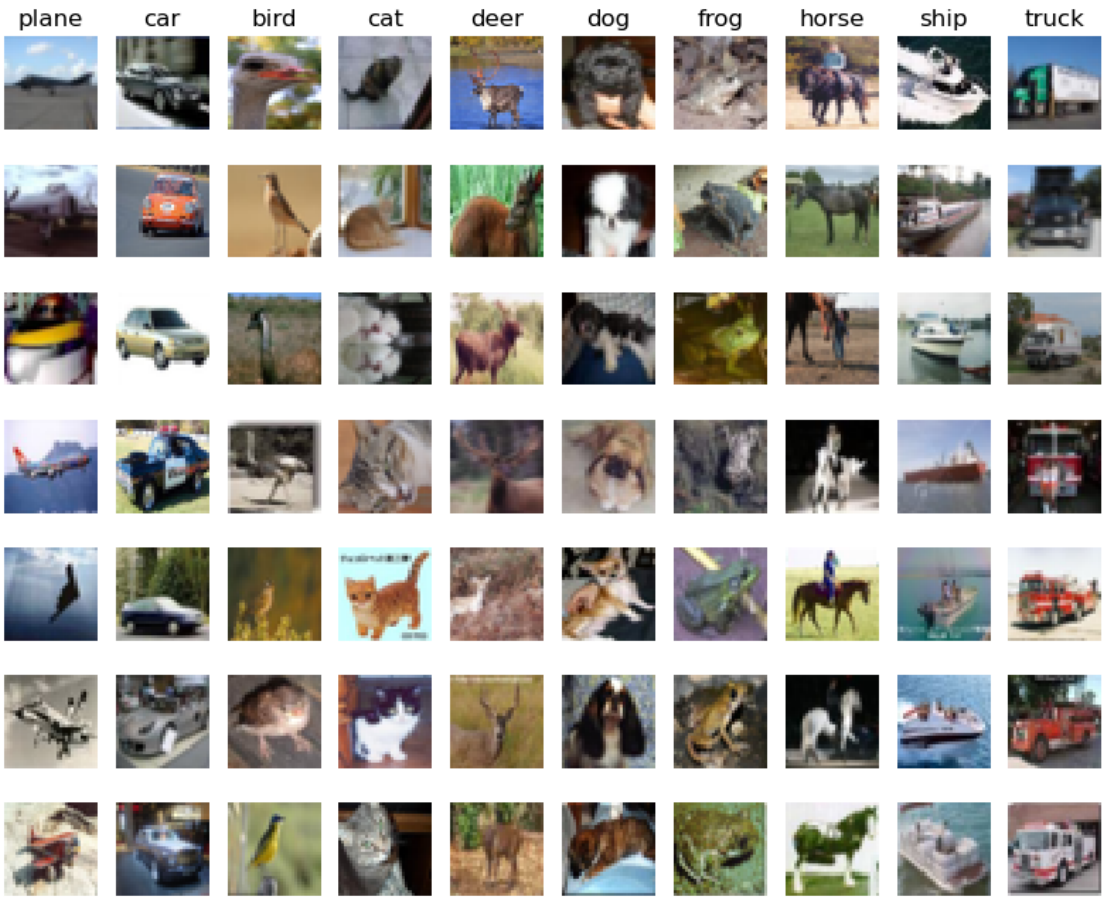
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [13]:
```
        # Load the raw CIFAR-10 data.
        cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

In [14]:
```
        # 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', 'tru
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 [15]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
```

```
y_train = y_train[mask]

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

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

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

```
In [17]: from cs231n.classifiers import KNearestNeighbor

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

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

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

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

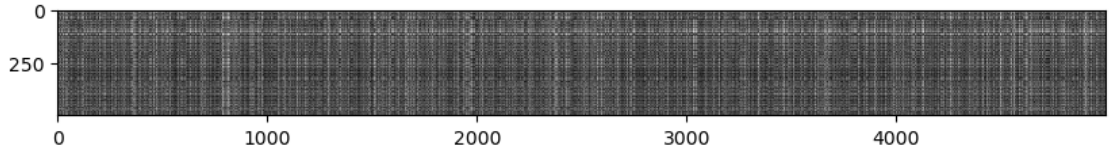
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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

```
In [18]: # Open cs231n/classifiers/k_nearest_neighbor.py and implement
         # compute_distances_two_loops.

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

```
(500, 5000)
```

In [19]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

Notice the structured patterns in the distance matrix, where some rows or columns are visibly brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

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

Your Answer : f1. For rows we consider the test data and for columns we consider the train data. So, the bright rows indicate that the test data is very different from the train data. 2. The bright columns indicate that the train data is very different from the test data..

In [20]:
```
# 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 [21]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. To clarify, both training and test examples are preprocessed in the same way.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

YourAnswer : 1. subtracting the mean will not effect the performance of the classifier. 2. Rotating the coordinate axes of the data will not change the performance of the classifier.

YourExplanation : Subtracting the mean - This will not change the L1 distance Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed

In [22]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

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

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

In [23]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

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

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

In [24]:
```
# 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 exec
    """
    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 implement

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

```
Two loop version took 18.762009 seconds
One loop version took 20.374184 seconds
No loop version took 0.172048 seconds
```

1.0.1 Cross-validation

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

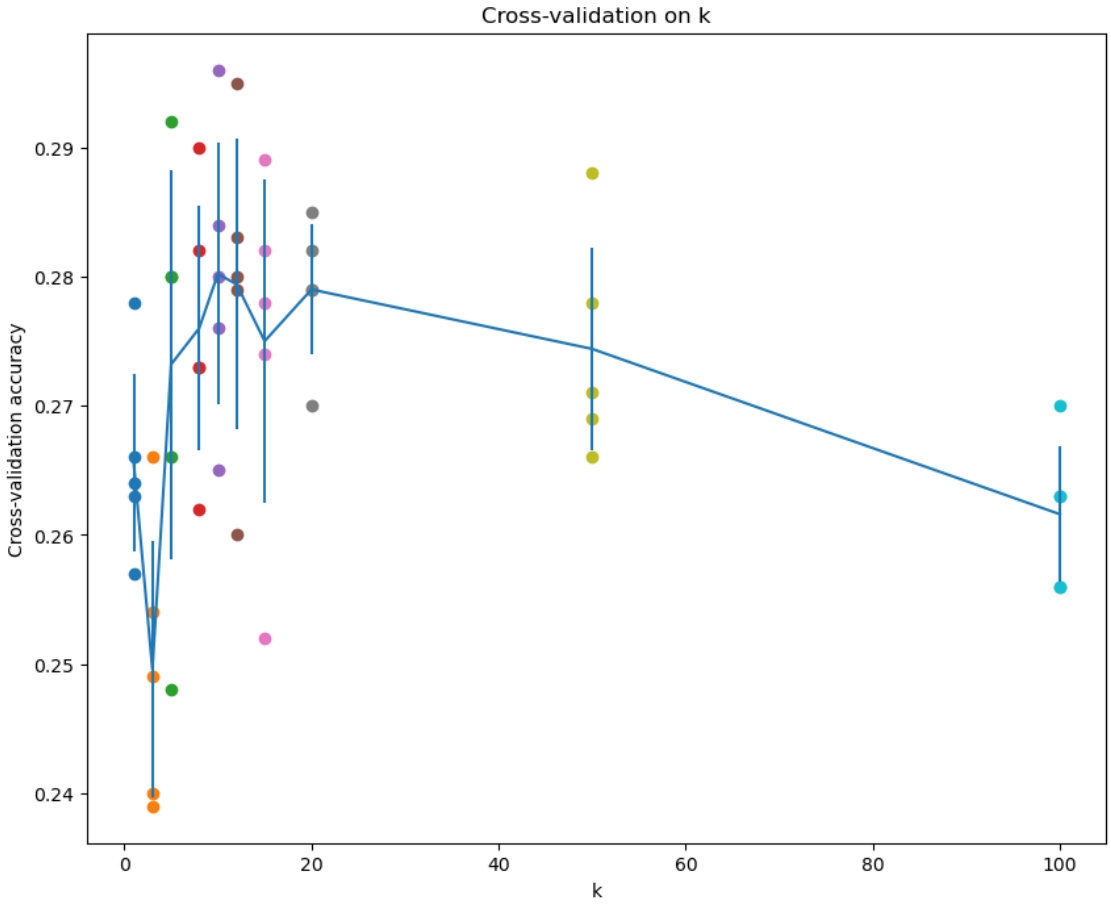
In [25]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
pass

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

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


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for k in k_choices:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        X_train_fold = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
        y_train_fold = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])
        X_test_fold = X_train_folds[i]
        y_test_fold = y_train_folds[i]
        classifier.train(X_train_fold, y_train_fold)
        y_test_pred_fold = classifier.predict(X_test_fold, k=k)
```

```
                num_correct = np.sum(y_test_pred_fold == y_test_fold)
                accuracy = float(num_correct) / len(y_test_fold)
                k_to_accuracies[k].append(accuracy)
        pass

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

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

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

```
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

In [26]:
```
# 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 [28]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear.

2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

YourAnswer :

YourExplanation :

svm

September 30, 2023

In [ ]:
```
# 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. 'cs231n/assignments/assignment1/'
FOLDERNAME = None
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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

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

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

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

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

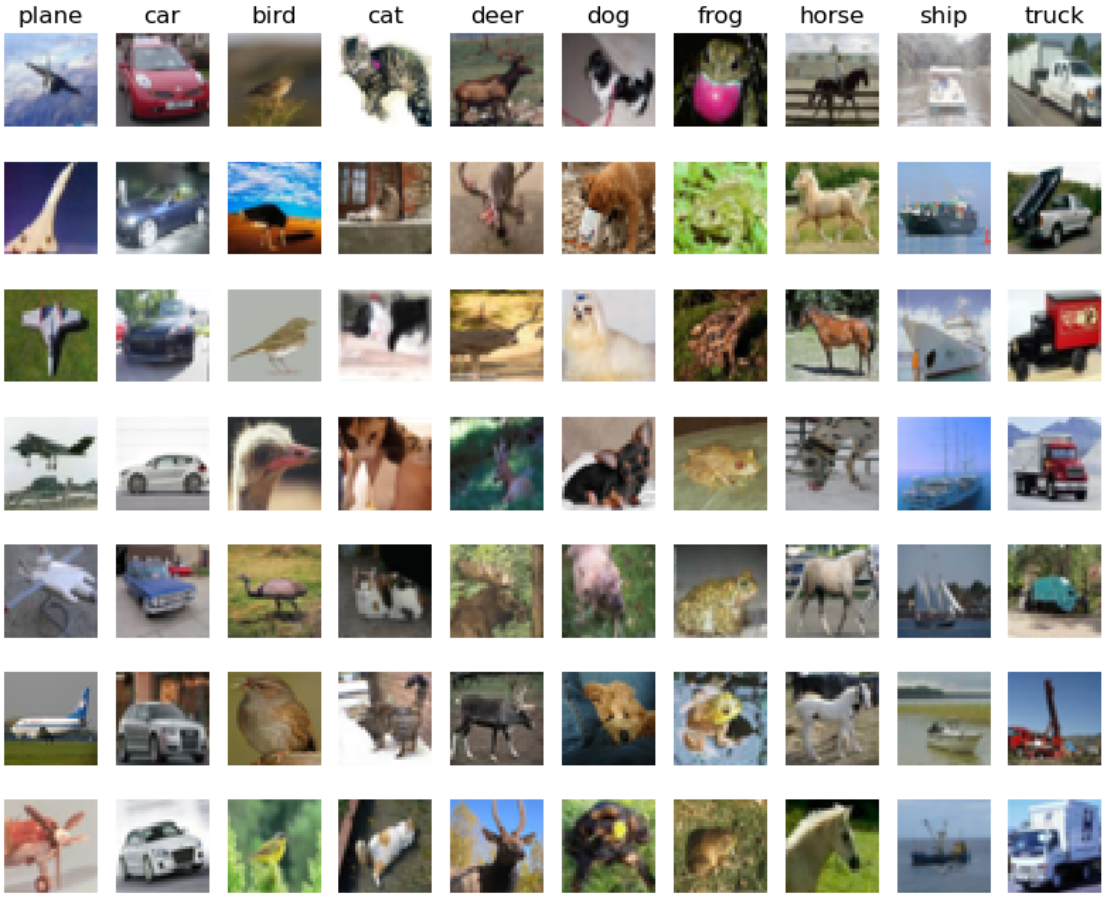
1.1 CIFAR-10 Data Loading and Preprocessing

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

# Cleaning up variables to prevent loading data multiple times (which may cause memory
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 [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
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 [4]:
```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500
```

```
        # Our validation set will be num_validation points from the original
        # training set.
        mask = range(num_training, num_training + num_validation)
        X_val = X_train[mask]
        y_val = y_train[mask]

        # Our training set will be the first num_train points from the original
        # training set.
        mask = range(num_training)
        X_train = X_train[mask]
        y_train = y_train[mask]

        # We will also make a development set, which is a small subset of
        # the training set.
        mask = np.random.choice(num_training, num_dev, replace=False)
        X_dev = X_train[mask]
        y_dev = y_train[mask]

        # We use the first num_test points of the original test set as our
        # test set.
        mask = range(num_test)
        X_test = X_test[mask]
        y_test = y_test[mask]

        print('Train data shape: ', X_train.shape)
        print('Train labels shape: ', y_train.shape)
        print('Validation data shape: ', X_val.shape)
        print('Validation labels shape: ', y_val.shape)
        print('Test data shape: ', X_test.shape)
        print('Test labels shape: ', y_test.shape)
```

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

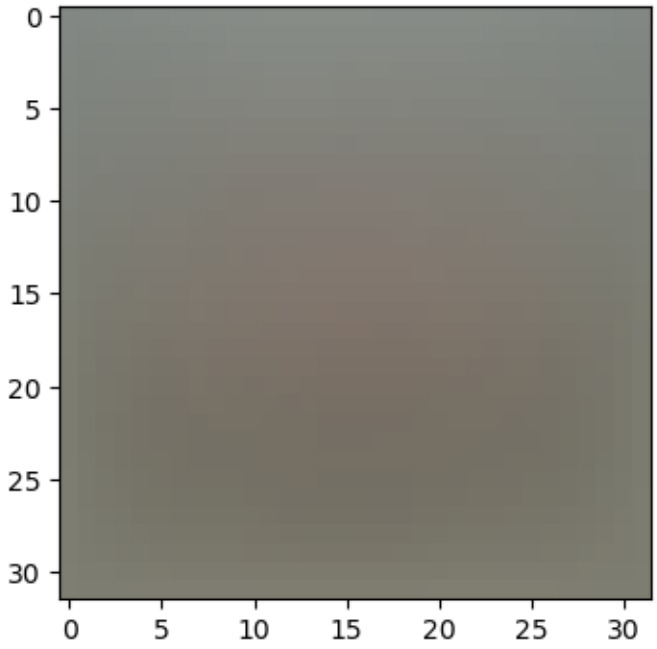
In [5]:
```
        # Preprocessing: reshape the image data into rows
        X_train = np.reshape(X_train, (X_train.shape[0], -1))
        X_val = np.reshape(X_val, (X_val.shape[0], -1))
        X_test = np.reshape(X_test, (X_test.shape[0], -1))
        X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

        # As a sanity check, print out the shapes of the data
        print('Training data shape: ', X_train.shape)
```

```
        print('Validation data shape: ', X_val.shape)
        print('Test data shape: ', X_test.shape)
        print('dev data shape: ', X_dev.shape)
```

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

In [6]:
```
        # Preprocessing: subtract the mean image
        # first: compute the image mean based on the training data
        mean_image = np.mean(X_train, axis=0)
        print(mean_image[:10]) # print a few of the elements
        plt.figure(figsize=(4,4))
        plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
        plt.show()

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

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

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

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




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

1.2 SVM Classifier

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

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

```
In [7]: # Evaluate the naive implementation of the loss we provided for you:
        from cs231n.classifiers.linear_svm import svm_loss_naive
        import time

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

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

```
loss: 9.491595
```

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

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

In [8]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

```
numerical: -11.494521 analytic: -11.494521, relative error: 2.972489e-12
numerical: 17.615242 analytic: 17.615242, relative error: 1.754137e-11
numerical: 1.130636 analytic: 1.130636, relative error: 1.757581e-10
numerical: -10.081741 analytic: -10.081741, relative error: 1.563206e-12
numerical: 3.371439 analytic: 3.371439, relative error: 8.130668e-11
numerical: 31.874432 analytic: 31.874432, relative error: 1.200348e-11
numerical: 2.831348 analytic: 2.831348, relative error: 3.749324e-11
numerical: -3.665822 analytic: -3.665822, relative error: 3.441029e-11
numerical: -8.045935 analytic: -8.045935, relative error: 9.942909e-12
numerical: 17.680131 analytic: 17.680131, relative error: 7.997867e-12
numerical: -8.035283 analytic: -8.035283, relative error: 1.468565e-11
numerical: -10.294367 analytic: -10.294367, relative error: 1.015295e-11
numerical: 6.939713 analytic: 6.939713, relative error: 1.235944e-11
numerical: 3.350620 analytic: 3.350620, relative error: 2.487590e-10
numerical: -39.739070 analytic: -39.739070, relative error: 7.012617e-12
numerical: -12.798615 analytic: -12.798615, relative error: 2.313664e-11
numerical: -21.007112 analytic: -21.007112, relative error: 9.139729e-12
numerical: 7.422796 analytic: 7.422796, relative error: 1.031634e-11
numerical: 6.488305 analytic: 6.488305, relative error: 1.825512e-11
numerical: -11.818074 analytic: -11.818074, relative error: 3.567572e-12
```

Inline Question 1

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in

one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer : fill this in.

Yes it is possible. SVM loss function, are not strictly differentiable at all points. The gradient check could fail if the function is not differentiable at the point where the gradient is being checked. The frequency of this happening would increase if the margin is decreased. gradient of the absolute value function is undefined at x = 0

In [9]:
```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
Naive loss: 9.491595e+00 computed in 0.037592s
Vectorized loss: 9.491595e+00 computed in 0.003113s
difference: -0.000000
```

In [11]:
```
# 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 0.041282s
Vectorized loss and gradient: computed in 0.003192s
difference: 0.000000
```

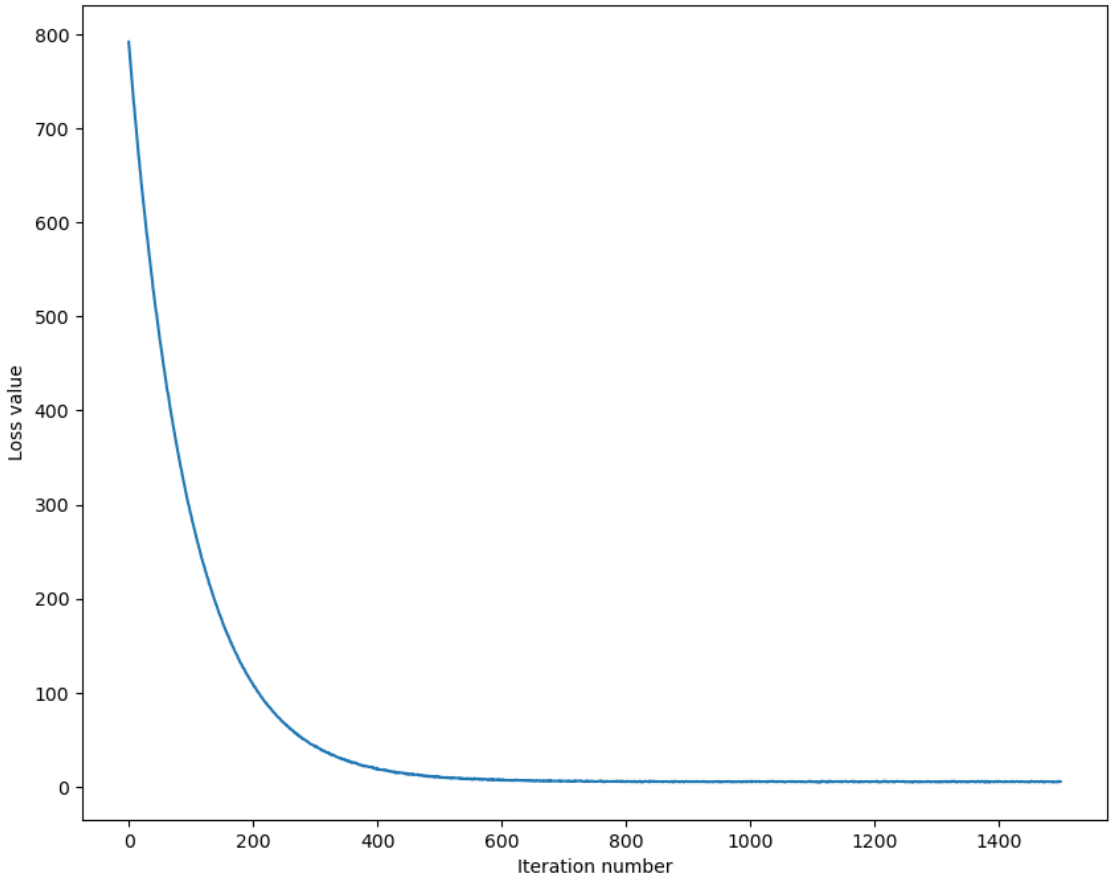
1.2.1 Stochastic Gradient Descent

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

```
In [12]: # In the file linear_classifier.py, implement SGD in the function
         # LinearClassifier.train() and then run it with the code below.
         from cs231n.classifiers import LinearSVM
         svm = LinearSVM()
         tic = time.time()
         loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                               num_iters=1500, verbose=True)
         toc = time.time()
         print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 791.898102
iteration 100 / 1500: loss 289.608982
iteration 200 / 1500: loss 108.703546
iteration 300 / 1500: loss 43.105771
iteration 400 / 1500: loss 18.690957
iteration 500 / 1500: loss 9.998584
iteration 600 / 1500: loss 7.401702
iteration 700 / 1500: loss 5.993777
iteration 800 / 1500: loss 5.433813
iteration 900 / 1500: loss 5.719031
iteration 1000 / 1500: loss 5.146404
iteration 1100 / 1500: loss 5.111731
iteration 1200 / 1500: loss 5.171397
iteration 1300 / 1500: loss 5.508725
iteration 1400 / 1500: loss 5.795730
That took 1.737391s
```

```
In [13]: # 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 [14]:
```
# 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: 0.367571
validation accuracy: 0.371000
```

In [15]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) on the validation set.

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

# results is dictionary mapping tuples of the form
```

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

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

# Provided as a reference. You may or may not want to change these hyperparameters
learning_rates = np.logspace(-7, -4,10)
regularization_strengths = np.logspace(1, 5,10)

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
results_array = np.zeros((regularization_strengths.shape[0], learning_rates.shape[0]))
for lr in learning_rates:
    tic = time.time()
    for rs in regularization_strengths:
        svm_test = LinearSVM()
        svm_test.train(X_train, y_train, learning_rate=lr, reg=rs, num_iters=600)
        train_acc = np.mean(svm_test.predict(X_train) == y_train)
        val_acc = np.mean(svm_test.predict(X_val) == y_val)
        results[(lr, rs)] = (train_acc, val_acc)
        results_array[np.where(regularization_strengths == rs)[0], np.where(learning_r
        if best_val < val_acc:
            best_val = val_acc
            best_svm = svm_test
    toc = time.time()


pass

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

# Print out results.
```

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

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

```
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/linear_svm.py:87: RuntimeWarning
  loss = np.sum(margins) / num_train + reg * np.sum(W * W)
/home/cooper/anaconda3/envs/comp4471/lib/python3.7/site-packages/numpy/core/fromnumeric.py:86:
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/linear_svm.py:87: RuntimeWarning
  loss = np.sum(margins) / num_train + reg * np.sum(W * W)
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/linear_svm.py:102: RuntimeWarni
  dW = (margins @ X).T / num_train + 2 * reg * W
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/linear_svm.py:83: RuntimeWarning
  scores = W.T @ X.T
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/linear_svm.py:83: RuntimeWarning
  scores = W.T @ X.T
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/linear_svm.py:85: RuntimeWarning
  margins = np.maximum(0, scores - correct_class_scores + 1)
```

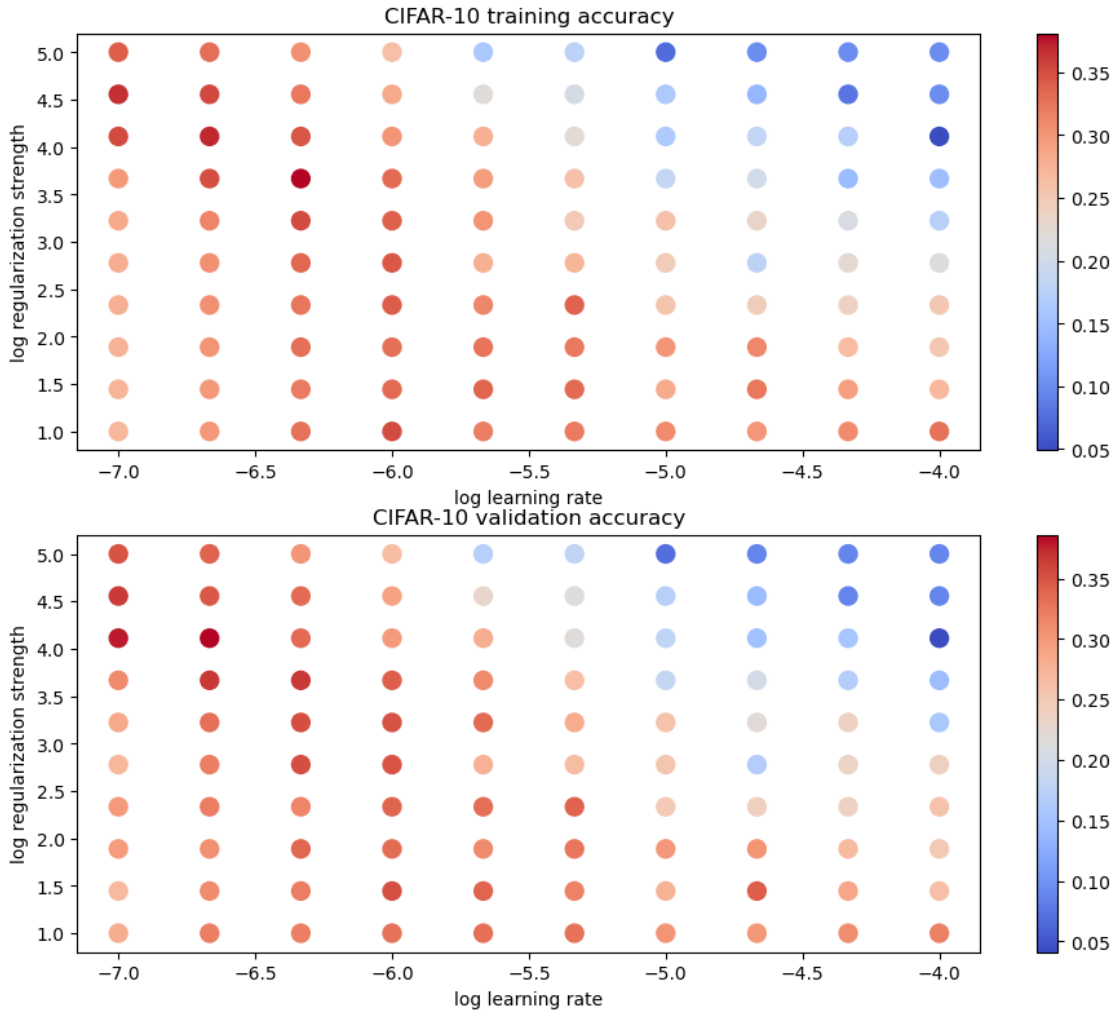
```
lr 1.000000e-07 reg 1.000000e+01 train accuracy: 0.271653 val accuracy: 0.282000
lr 1.000000e-07 reg 2.782559e+01 train accuracy: 0.273531 val accuracy: 0.268000
lr 1.000000e-07 reg 7.742637e+01 train accuracy: 0.276367 val accuracy: 0.298000
lr 1.000000e-07 reg 2.154435e+02 train accuracy: 0.279857 val accuracy: 0.299000
lr 1.000000e-07 reg 5.994843e+02 train accuracy: 0.280102 val accuracy: 0.272000
lr 1.000000e-07 reg 1.668101e+03 train accuracy: 0.284429 val accuracy: 0.285000
lr 1.000000e-07 reg 4.641589e+03 train accuracy: 0.297429 val accuracy: 0.312000
lr 1.000000e-07 reg 1.291550e+04 train accuracy: 0.352224 val accuracy: 0.379000
lr 1.000000e-07 reg 3.593814e+04 train accuracy: 0.366143 val accuracy: 0.365000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.340490 val accuracy: 0.350000
lr 2.154435e-07 reg 1.000000e+01 train accuracy: 0.299163 val accuracy: 0.320000
lr 2.154435e-07 reg 2.782559e+01 train accuracy: 0.298510 val accuracy: 0.311000
lr 2.154435e-07 reg 7.742637e+01 train accuracy: 0.302082 val accuracy: 0.307000
lr 2.154435e-07 reg 2.154435e+02 train accuracy: 0.305673 val accuracy: 0.322000
lr 2.154435e-07 reg 5.994843e+02 train accuracy: 0.305327 val accuracy: 0.320000
lr 2.154435e-07 reg 1.668101e+03 train accuracy: 0.314082 val accuracy: 0.332000
lr 2.154435e-07 reg 4.641589e+03 train accuracy: 0.350612 val accuracy: 0.366000
lr 2.154435e-07 reg 1.291550e+04 train accuracy: 0.369592 val accuracy: 0.386000
lr 2.154435e-07 reg 3.593814e+04 train accuracy: 0.353592 val accuracy: 0.348000
lr 2.154435e-07 reg 1.000000e+05 train accuracy: 0.329653 val accuracy: 0.341000
lr 4.641589e-07 reg 1.000000e+01 train accuracy: 0.326776 val accuracy: 0.320000
lr 4.641589e-07 reg 2.782559e+01 train accuracy: 0.320633 val accuracy: 0.322000
lr 4.641589e-07 reg 7.742637e+01 train accuracy: 0.328449 val accuracy: 0.338000
lr 4.641589e-07 reg 2.154435e+02 train accuracy: 0.324592 val accuracy: 0.316000
lr 4.641589e-07 reg 5.994843e+02 train accuracy: 0.335571 val accuracy: 0.354000
```

```
lr 4.641589e-07 reg 1.668101e+03 train accuracy: 0.351898 val accuracy: 0.354000
lr 4.641589e-07 reg 4.641589e+03 train accuracy: 0.381204 val accuracy: 0.366000
lr 4.641589e-07 reg 1.291550e+04 train accuracy: 0.345449 val accuracy: 0.336000
lr 4.641589e-07 reg 3.593814e+04 train accuracy: 0.323490 val accuracy: 0.335000
lr 4.641589e-07 reg 1.000000e+05 train accuracy: 0.304796 val accuracy: 0.303000
lr 1.000000e-06 reg 1.000000e+01 train accuracy: 0.351980 val accuracy: 0.330000
lr 1.000000e-06 reg 2.782559e+01 train accuracy: 0.333653 val accuracy: 0.355000
lr 1.000000e-06 reg 7.742637e+01 train accuracy: 0.328490 val accuracy: 0.334000
lr 1.000000e-06 reg 2.154435e+02 train accuracy: 0.341959 val accuracy: 0.339000
lr 1.000000e-06 reg 5.994843e+02 train accuracy: 0.344061 val accuracy: 0.350000
lr 1.000000e-06 reg 1.668101e+03 train accuracy: 0.338837 val accuracy: 0.352000
lr 1.000000e-06 reg 4.641589e+03 train accuracy: 0.333000 val accuracy: 0.343000
lr 1.000000e-06 reg 1.291550e+04 train accuracy: 0.301959 val accuracy: 0.299000
lr 1.000000e-06 reg 3.593814e+04 train accuracy: 0.284224 val accuracy: 0.293000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.260816 val accuracy: 0.266000
lr 2.154435e-06 reg 1.000000e+01 train accuracy: 0.318490 val accuracy: 0.332000
lr 2.154435e-06 reg 2.782559e+01 train accuracy: 0.336612 val accuracy: 0.339000
lr 2.154435e-06 reg 7.742637e+01 train accuracy: 0.325592 val accuracy: 0.312000
lr 2.154435e-06 reg 2.154435e+02 train accuracy: 0.313571 val accuracy: 0.333000
lr 2.154435e-06 reg 5.994843e+02 train accuracy: 0.278653 val accuracy: 0.277000
lr 2.154435e-06 reg 1.668101e+03 train accuracy: 0.302653 val accuracy: 0.336000
lr 2.154435e-06 reg 4.641589e+03 train accuracy: 0.296204 val accuracy: 0.313000
lr 2.154435e-06 reg 1.291550e+04 train accuracy: 0.277898 val accuracy: 0.281000
lr 2.154435e-06 reg 3.593814e+04 train accuracy: 0.218204 val accuracy: 0.231000
lr 2.154435e-06 reg 1.000000e+05 train accuracy: 0.161490 val accuracy: 0.170000
lr 4.641589e-06 reg 1.000000e+01 train accuracy: 0.320286 val accuracy: 0.330000
lr 4.641589e-06 reg 2.782559e+01 train accuracy: 0.332122 val accuracy: 0.318000
lr 4.641589e-06 reg 7.742637e+01 train accuracy: 0.321857 val accuracy: 0.328000
lr 4.641589e-06 reg 2.154435e+02 train accuracy: 0.338367 val accuracy: 0.341000
lr 4.641589e-06 reg 5.994843e+02 train accuracy: 0.272245 val accuracy: 0.266000
lr 4.641589e-06 reg 1.668101e+03 train accuracy: 0.249571 val accuracy: 0.283000
lr 4.641589e-06 reg 4.641589e+03 train accuracy: 0.260061 val accuracy: 0.263000
lr 4.641589e-06 reg 1.291550e+04 train accuracy: 0.222776 val accuracy: 0.217000
lr 4.641589e-06 reg 3.593814e+04 train accuracy: 0.205857 val accuracy: 0.215000
lr 4.641589e-06 reg 1.000000e+05 train accuracy: 0.178204 val accuracy: 0.180000
lr 1.000000e-05 reg 1.000000e+01 train accuracy: 0.309020 val accuracy: 0.305000
lr 1.000000e-05 reg 2.782559e+01 train accuracy: 0.283796 val accuracy: 0.276000
lr 1.000000e-05 reg 7.742637e+01 train accuracy: 0.300939 val accuracy: 0.302000
lr 1.000000e-05 reg 2.154435e+02 train accuracy: 0.256224 val accuracy: 0.249000
lr 1.000000e-05 reg 5.994843e+02 train accuracy: 0.247408 val accuracy: 0.255000
lr 1.000000e-05 reg 1.668101e+03 train accuracy: 0.259796 val accuracy: 0.258000
lr 1.000000e-05 reg 4.641589e+03 train accuracy: 0.187796 val accuracy: 0.185000
lr 1.000000e-05 reg 1.291550e+04 train accuracy: 0.165102 val accuracy: 0.179000
lr 1.000000e-05 reg 3.593814e+04 train accuracy: 0.163592 val accuracy: 0.170000
lr 1.000000e-05 reg 1.000000e+05 train accuracy: 0.072388 val accuracy: 0.069000
lr 2.154435e-05 reg 1.000000e+01 train accuracy: 0.300306 val accuracy: 0.300000
lr 2.154435e-05 reg 2.782559e+01 train accuracy: 0.323449 val accuracy: 0.344000
lr 2.154435e-05 reg 7.742637e+01 train accuracy: 0.311245 val accuracy: 0.305000
```

```
lr 2.154435e-05 reg 2.154435e+02 train accuracy: 0.243959 val accuracy: 0.241000
lr 2.154435e-05 reg 5.994843e+02 train accuracy: 0.179122 val accuracy: 0.167000
lr 2.154435e-05 reg 1.668101e+03 train accuracy: 0.233551 val accuracy: 0.219000
lr 2.154435e-05 reg 4.641589e+03 train accuracy: 0.202082 val accuracy: 0.201000
lr 2.154435e-05 reg 1.291550e+04 train accuracy: 0.184796 val accuracy: 0.148000
lr 2.154435e-05 reg 3.593814e+04 train accuracy: 0.140816 val accuracy: 0.141000
lr 2.154435e-05 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-05 reg 1.000000e+01 train accuracy: 0.309143 val accuracy: 0.310000
lr 4.641589e-05 reg 2.782559e+01 train accuracy: 0.294122 val accuracy: 0.288000
lr 4.641589e-05 reg 7.742637e+01 train accuracy: 0.266408 val accuracy: 0.268000
lr 4.641589e-05 reg 2.154435e+02 train accuracy: 0.237429 val accuracy: 0.239000
lr 4.641589e-05 reg 5.994843e+02 train accuracy: 0.226612 val accuracy: 0.235000
lr 4.641589e-05 reg 1.668101e+03 train accuracy: 0.209837 val accuracy: 0.239000
lr 4.641589e-05 reg 4.641589e+03 train accuracy: 0.146306 val accuracy: 0.167000
lr 4.641589e-05 reg 1.291550e+04 train accuracy: 0.173224 val accuracy: 0.153000
lr 4.641589e-05 reg 3.593814e+04 train accuracy: 0.080224 val accuracy: 0.088000
lr 4.641589e-05 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 1.000000e+01 train accuracy: 0.327082 val accuracy: 0.319000
lr 1.000000e-04 reg 2.782559e+01 train accuracy: 0.270571 val accuracy: 0.262000
lr 1.000000e-04 reg 7.742637e+01 train accuracy: 0.253265 val accuracy: 0.251000
lr 1.000000e-04 reg 2.154435e+02 train accuracy: 0.253388 val accuracy: 0.258000
lr 1.000000e-04 reg 5.994843e+02 train accuracy: 0.216000 val accuracy: 0.240000
lr 1.000000e-04 reg 1.668101e+03 train accuracy: 0.173857 val accuracy: 0.158000
lr 1.000000e-04 reg 4.641589e+03 train accuracy: 0.148143 val accuracy: 0.143000
lr 1.000000e-04 reg 1.291550e+04 train accuracy: 0.049000 val accuracy: 0.041000
lr 1.000000e-04 reg 3.593814e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.386000
```

In [16]:
```
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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

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

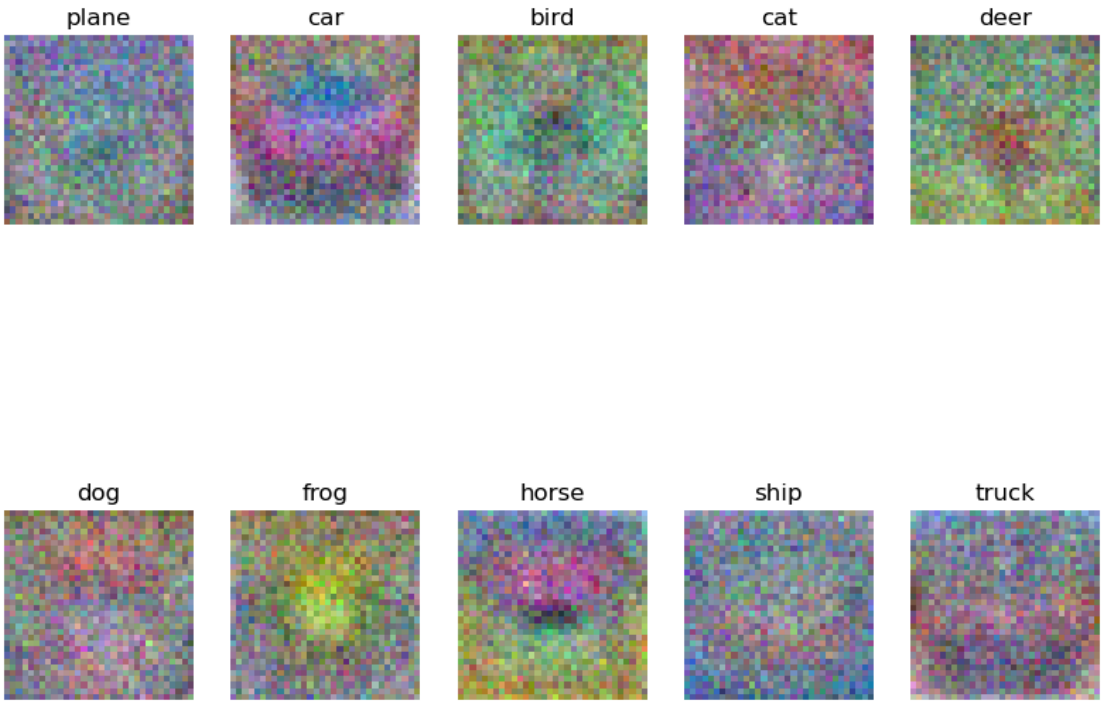
```
plt.title('CIFAR-10 training accuracy')

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



In [17]:
```
# 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: 0.358000
```

```
In [19]: # Visualize the learned weights for each class.
         # Depending on your choice of learning rate and regularization strength, these may
         # or may not be nice to look at.
         w = best_svm.W[:-1,:] # strip out the bias
         w = w.reshape(32, 32, 3, 10)
         w_min, w_max = np.min(w), np.max(w)
         classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
         for i in range(10):
             plt.subplot(2, 5, i + 1)

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



Inline question 2

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

Your Answer : fill this in

The visualized SVM weights look like the average of all the images in the dataset. This is because the SVM loss function is the sum of the loss of each image in the dataset. SVM model may have a different point of view in terms of identifying images from us human.

softmax

September 30, 2023

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. 'cs231n/assignments/assignment1/'
FOLDERNAME = None
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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)

<ipython-input-1-189b17b10db6> in <module>
  1 # This mounts your Google Drive to the Colab VM.
----> 2 from google.colab import drive
  3 drive.mount('/content/drive')
  4
  5 # TODO: Enter the foldername in your Drive where you have saved the unzipped


ModuleNotFoundError: No module named 'google'
```

1 Softmax exercise

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

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

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

In [5]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

In [6]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=5
    """
    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 = 'cs231n/datasets/cifar-10-batches-py'

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

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

    # subsample the data
```

```
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
```

```
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

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

```
In [7]: # First implement the naive softmax loss function with nested loops.
        # Open the file cs231n/classifiers/softmax.py and implement the
        # softmax_loss_naive function.

        from cs231n.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: 2.378504
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : Fill this in

we should have the same score for all classes no matter the input. Therefore, we should expect the loss starts from N=10.

```
In [8]: # 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 cs231n.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: 0.091586 analytic: 0.091586, relative error: 4.778132e-07
numerical: 0.635370 analytic: 0.635370, relative error: 1.018943e-08
numerical: 0.139359 analytic: 0.139359, relative error: 4.266308e-07
numerical: 1.181014 analytic: 1.181014, relative error: 3.471331e-08
numerical: 3.243992 analytic: 3.243992, relative error: 4.082824e-09
numerical: -1.884724 analytic: -1.884724, relative error: 2.323749e-08
numerical: 0.400701 analytic: 0.400701, relative error: 7.376342e-08
numerical: -1.469843 analytic: -1.469843, relative error: 4.081758e-09
numerical: -3.895666 analytic: -3.895665, relative error: 1.621798e-08
numerical: 0.786447 analytic: 0.786447, relative error: 1.113026e-07
numerical: 1.090752 analytic: 1.090752, relative error: 7.989243e-08
numerical: -0.818332 analytic: -0.818332, relative error: 1.133686e-08
numerical: 1.425177 analytic: 1.425177, relative error: 3.117591e-08
numerical: 1.605318 analytic: 1.605318, relative error: 5.079022e-09
numerical: 0.398720 analytic: 0.398720, relative error: 1.552745e-07
numerical: -0.142474 analytic: -0.142474, relative error: 3.811365e-08
numerical: -0.370396 analytic: -0.370396, relative error: 1.165182e-07
numerical: -0.313881 analytic: -0.313881, relative error: 7.199029e-08
numerical: -1.178505 analytic: -1.178505, relative error: 1.965704e-08
numerical: -0.739432 analytic: -0.739432, relative error: 9.704426e-08
```

In [9]:
```
        # Now that we have a naive implementation of the softmax loss function and its gradien
        # implement a vectorized version in softmax_loss_vectorized.
        # The two versions should compute the same results, but the vectorized version should
        # 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 cs231n.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: 2.378504e+00 computed in 0.050538s
vectorized loss: 2.378504e+00 computed in 0.062936s
```

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

In [10]:
```
# 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 cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

# Provided as a reference. You may or may not want to change these hyperparameters
learning_rates = np.logspace(-1,-7,10)
regularization_strengths = np.logspace(1,4,10)

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
results_array = np.zeros((regularization_strengths.shape[0], learning_rates.shape[0]))
for lr in learning_rates:
    tic = time.time()
    for rs in regularization_strengths:
        softm = Softmax()
        softm.train(X_train, y_train, learning_rate=lr, reg=rs, num_iters=300)
        train_acc = np.mean(softm.predict(X_train) == y_train)
        val_acc = np.mean(softm.predict(X_val) == y_val)
        results[(lr, rs)] = (train_acc, val_acc)
        results_array[np.where(regularization_strengths == rs)[0], np.where(learning_r
        if best_val < val_acc:
            best_val = val_acc
            best_softmax = softm
    toc = time.time()
    print("Finish %d/%d, time for this iter is %fs" % (np.where(learning_rates == lr)

pass

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

# Print out results.
for lr, reg in sorted(results):
```

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

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

```
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/softmax.py:78: RuntimeWarning: o
  exp_Wx = np.exp(Wx)
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/softmax.py:80: RuntimeWarning: i
  loss = np.sum(-np.log(exp_Wx[np.arange(num_train), y] / sum_exp_Wx))
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/softmax.py:80: RuntimeWarning: o
  loss = np.sum(-np.log(exp_Wx[np.arange(num_train), y] / sum_exp_Wx))
/home/cooper/Documents/comp4471/assignment1/cs231n/classifiers/softmax.py:83: RuntimeWarning: i
  weights = exp_Wx / sum_exp_Wx.reshape(-1, 1)
```

```
Finish 1/10, time for this iter is 3.846262s
Finish 2/10, time for this iter is 3.219445s
Finish 3/10, time for this iter is 2.940309s
Finish 4/10, time for this iter is 3.171849s
Finish 5/10, time for this iter is 5.128260s
Finish 6/10, time for this iter is 5.014900s
Finish 7/10, time for this iter is 4.161523s
Finish 8/10, time for this iter is 7.821477s
Finish 9/10, time for this iter is 3.413399s
Finish 10/10, time for this iter is 5.294451s
lr 1.000000e-07 reg 1.000000e+01 train accuracy: 0.189633 val accuracy: 0.191000
lr 1.000000e-07 reg 2.154435e+01 train accuracy: 0.194755 val accuracy: 0.173000
lr 1.000000e-07 reg 4.641589e+01 train accuracy: 0.179408 val accuracy: 0.181000
lr 1.000000e-07 reg 1.000000e+02 train accuracy: 0.178388 val accuracy: 0.181000
lr 1.000000e-07 reg 2.154435e+02 train accuracy: 0.189755 val accuracy: 0.202000
lr 1.000000e-07 reg 4.641589e+02 train accuracy: 0.183571 val accuracy: 0.192000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.169367 val accuracy: 0.183000
lr 1.000000e-07 reg 2.154435e+03 train accuracy: 0.185857 val accuracy: 0.200000
lr 1.000000e-07 reg 4.641589e+03 train accuracy: 0.187286 val accuracy: 0.188000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.206000 val accuracy: 0.196000
lr 4.641589e-07 reg 1.000000e+01 train accuracy: 0.242959 val accuracy: 0.237000
lr 4.641589e-07 reg 2.154435e+01 train accuracy: 0.243041 val accuracy: 0.252000
lr 4.641589e-07 reg 4.641589e+01 train accuracy: 0.238612 val accuracy: 0.257000
lr 4.641589e-07 reg 1.000000e+02 train accuracy: 0.255633 val accuracy: 0.267000
lr 4.641589e-07 reg 2.154435e+02 train accuracy: 0.250102 val accuracy: 0.248000
lr 4.641589e-07 reg 4.641589e+02 train accuracy: 0.258714 val accuracy: 0.261000
lr 4.641589e-07 reg 1.000000e+03 train accuracy: 0.257878 val accuracy: 0.270000
lr 4.641589e-07 reg 2.154435e+03 train accuracy: 0.281184 val accuracy: 0.286000
lr 4.641589e-07 reg 4.641589e+03 train accuracy: 0.317714 val accuracy: 0.324000
lr 4.641589e-07 reg 1.000000e+04 train accuracy: 0.347551 val accuracy: 0.347000
lr 2.154435e-06 reg 1.000000e+01 train accuracy: 0.306367 val accuracy: 0.317000
lr 2.154435e-06 reg 2.154435e+01 train accuracy: 0.309571 val accuracy: 0.311000
```

```
lr 2.154435e-06 reg 4.641589e+01 train accuracy: 0.316265 val accuracy: 0.310000
lr 2.154435e-06 reg 1.000000e+02 train accuracy: 0.315306 val accuracy: 0.310000
lr 2.154435e-06 reg 2.154435e+02 train accuracy: 0.316959 val accuracy: 0.315000
lr 2.154435e-06 reg 4.641589e+02 train accuracy: 0.347143 val accuracy: 0.333000
lr 2.154435e-06 reg 1.000000e+03 train accuracy: 0.365143 val accuracy: 0.363000
lr 2.154435e-06 reg 2.154435e+03 train accuracy: 0.370633 val accuracy: 0.394000
lr 2.154435e-06 reg 4.641589e+03 train accuracy: 0.355367 val accuracy: 0.368000
lr 2.154435e-06 reg 1.000000e+04 train accuracy: 0.336531 val accuracy: 0.336000
lr 1.000000e-05 reg 1.000000e+01 train accuracy: 0.273755 val accuracy: 0.271000
lr 1.000000e-05 reg 2.154435e+01 train accuracy: 0.317245 val accuracy: 0.329000
lr 1.000000e-05 reg 4.641589e+01 train accuracy: 0.301306 val accuracy: 0.307000
lr 1.000000e-05 reg 1.000000e+02 train accuracy: 0.316163 val accuracy: 0.334000
lr 1.000000e-05 reg 2.154435e+02 train accuracy: 0.297612 val accuracy: 0.301000
lr 1.000000e-05 reg 4.641589e+02 train accuracy: 0.275265 val accuracy: 0.277000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.242796 val accuracy: 0.244000
lr 1.000000e-05 reg 2.154435e+03 train accuracy: 0.229490 val accuracy: 0.248000
lr 1.000000e-05 reg 4.641589e+03 train accuracy: 0.214408 val accuracy: 0.212000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.143367 val accuracy: 0.148000
lr 4.641589e-05 reg 1.000000e+01 train accuracy: 0.276224 val accuracy: 0.284000
lr 4.641589e-05 reg 2.154435e+01 train accuracy: 0.268918 val accuracy: 0.283000
lr 4.641589e-05 reg 4.641589e+01 train accuracy: 0.277816 val accuracy: 0.282000
lr 4.641589e-05 reg 1.000000e+02 train accuracy: 0.237551 val accuracy: 0.234000
lr 4.641589e-05 reg 2.154435e+02 train accuracy: 0.242776 val accuracy: 0.268000
lr 4.641589e-05 reg 4.641589e+02 train accuracy: 0.183122 val accuracy: 0.185000
lr 4.641589e-05 reg 1.000000e+03 train accuracy: 0.215959 val accuracy: 0.200000
lr 4.641589e-05 reg 2.154435e+03 train accuracy: 0.187408 val accuracy: 0.193000
lr 4.641589e-05 reg 4.641589e+03 train accuracy: 0.134429 val accuracy: 0.149000
lr 4.641589e-05 reg 1.000000e+04 train accuracy: 0.102184 val accuracy: 0.099000
lr 2.154435e-04 reg 1.000000e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 2.154435e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 4.641589e+01 train accuracy: 0.263000 val accuracy: 0.258000
lr 2.154435e-04 reg 1.000000e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 2.154435e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 4.641589e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 2.154435e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 4.641589e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-04 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 1.000000e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 2.154435e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 4.641589e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 1.000000e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 2.154435e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 4.641589e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 2.154435e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 4.641589e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-03 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
```

```
lr 4.641589e-03 reg 1.000000e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 2.154435e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 4.641589e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 1.000000e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 2.154435e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 4.641589e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 2.154435e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 4.641589e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.641589e-03 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 1.000000e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 2.154435e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 4.641589e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 1.000000e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 2.154435e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 4.641589e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 2.154435e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 4.641589e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.154435e-02 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 1.000000e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 2.154435e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 4.641589e+01 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 1.000000e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 2.154435e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 4.641589e+02 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 2.154435e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 4.641589e+03 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.394000
```

```
In [11]: # 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: 0.361000
```

Inline Question 2 - *True or False*

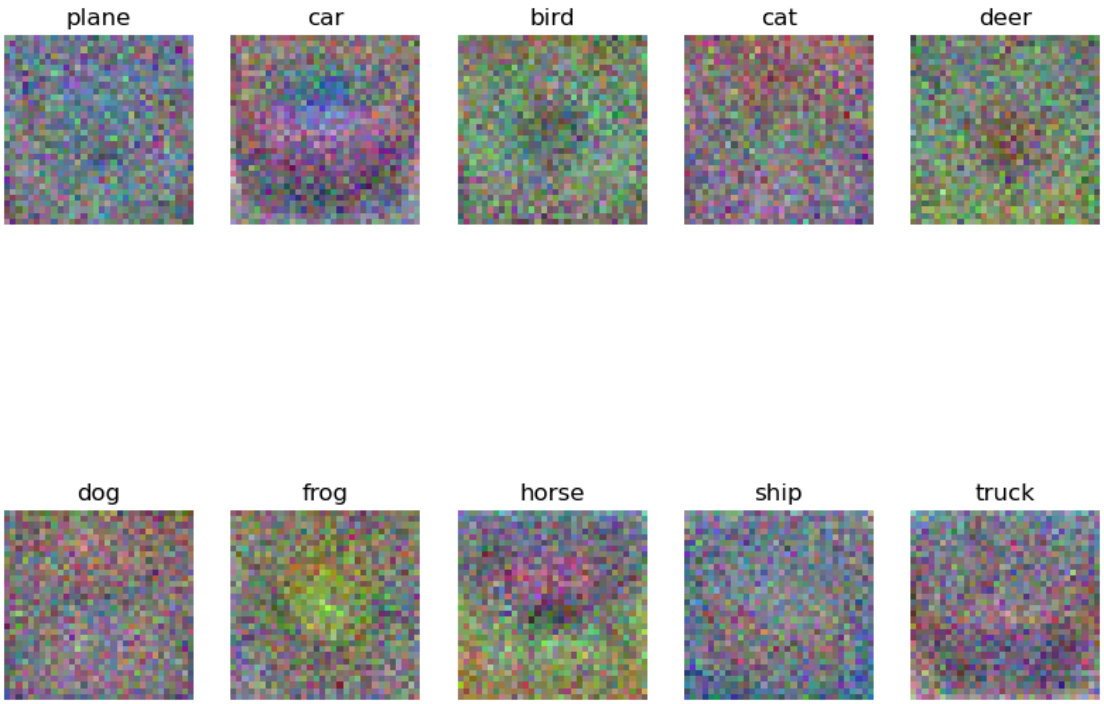
Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

YourAnswer : True

YourExplanation : Softmax classifier loss, the per-datapoint loss is defined as the negative log

probability assigned to the correct class.The probability for each class is computed by exponentiating the score for that class and normalizing over all classes.

In [12]:
```
# 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', 'tru
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 [ ]:

two_layer_net

September 30, 2023

```
In [ ]: # 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. 'cs231n/assignments/assignment1/'
        FOLDERNAME = None
        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/$FOLDERNAME/cs231n/datasets/
        !bash get_datasets.sh
        %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks using a modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients
```

```
  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

```
In [1]: # As usual, a bit of setup
        from __future__ import print_function
        import time
        import numpy as np
        import matplotlib.pyplot as plt
        from cs231n.classifiers.fc_net import *
        from cs231n.data_utils import get_CIFAR10_data
        from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arra
        from cs231n.solver import Solver

        %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))))

In [2]: # Load the (preprocessed) CIFAR10 data.
```

```
        data = get_CIFAR10_data()
        for k, v in list(data.items()):
          print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

2 Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

```
In [6]: # Test the affine_forward function

        num_inputs = 2
        input_shape = (4, 5, 6)
        output_dim = 3

        input_size = num_inputs * np.prod(input_shape)
        weight_size = output_dim * np.prod(input_shape)

        x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
        w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
        b = np.linspace(-0.3, 0.1, num=output_dim)

        out, _ = affine_forward(x, w, b)
        correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                                [ 3.25553199,  3.5141327,   3.77273342]])

        # Compare your output with ours. The error should be around e-9 or less.
        print('Testing affine_forward function:')
        print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.7698500479884e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

```
In [7]: # Test the affine_backward function
        np.random.seed(231)
        x = np.random.randn(10, 2, 3)
        w = np.random.randn(6, 5)
        b = np.random.randn(5)
        dout = np.random.randn(10, 5)

        dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
        dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dout)
        db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout)

        _, cache = affine_forward(x, w, b)
        dx, dw, db = affine_backward(dout, cache)

        # The error should be around e-10 or less
        print('Testing affine_backward function:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  1.0908199508708189e-10
dw error:  2.1752635504596857e-10
db error:  7.736978834487815e-12
```

4 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

```
In [8]: # Test the relu_forward function

        x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

        out, _ = relu_forward(x)
        correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                                [ 0.,          0.,          0.04545455,  0.13636364,],
                                [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

        # Compare your output with ours. The error should be on the order of e-8
        print('Testing relu_forward function:')
        print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

```
In [9]: np.random.seed(231)
        x = np.random.randn(10, 10)
        dout = np.random.randn(*x.shape)

        dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

        _, cache = relu_forward(x)
        dx = relu_backward(dout, cache)

        # The error should be on the order of e-12
        print('Testing relu_backward function:')
        print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. ReLU 3. Leaky ReLU

5.2 Answer:

[1]

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

```
In [10]: from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
         np.random.seed(231)
         x = np.random.randn(2, 3, 4)
         w = np.random.randn(12, 10)
         b = np.random.randn(10)
```

```
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x, d
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, d
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, d

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.395535042049294e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

You can make sure that the implementations are correct by running the following:

In [25]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be around the o
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be arou
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.483503037636722e-09
```

8 Two-layer network

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. Read through it to make sure you understand the API. You can run the cell below to test your implementation.

```
In [26]: np.random.seed(231)
         N, D, H, C = 3, 5, 50, 7
         X = np.random.randn(N, D)
         y = np.random.randint(C, size=N)

         std = 1e-3
         model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

         print('Testing initialization ... ')
         W1_std = abs(model.params['W1'].std() - std)
         b1 = model.params['b1']
         W2_std = abs(model.params['W2'].std() - std)
         b2 = model.params['b2']
         assert W1_std < std / 10, 'First layer weights do not seem right'
         assert np.all(b1 == 0), 'First layer biases do not seem right'
         assert W2_std < std / 10, 'Second layer weights do not seem right'
         assert np.all(b2 == 0), 'Second layer biases do not seem right'

         print('Testing test-time forward pass ... ')
         model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
         model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
         model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
         model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
         X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
         scores = model.loss(X)
         correct_scores = np.asarray(
           [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,
            [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,
            [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,
         scores_diff = np.abs(scores - correct_scores).sum()
         assert scores_diff < 1e-6, 'Problem with test-time forward pass'
```

```
print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-08
W2 relative error: 3.42e-10
b1 relative error: 6.55e-09
b2 relative error: 2.53e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 1.37e-07
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

9 Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about 36% accuracy on the validation set.

In [35]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None
```

```
        ################################################################################
        # TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
        # accuracy on the validation set.                                              #
        ################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        solver = Solver(model, data, update_rule="sgd",
                        optim_config={"learning_rate" : 4e-4}, lr_decay=0.99,
                        batch_size=200, print_every=100, num_epochs=10,)
        solver.train()
        pass

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

```
(Iteration 1 / 2450) loss: 2.302202
(Epoch 0 / 10) train acc: 0.109000; val_acc: 0.124000
(Iteration 101 / 2450) loss: 1.983173
(Iteration 201 / 2450) loss: 1.911570
(Epoch 1 / 10) train acc: 0.339000; val_acc: 0.365000
(Iteration 301 / 2450) loss: 1.823363
(Iteration 401 / 2450) loss: 1.723466
(Epoch 2 / 10) train acc: 0.407000; val_acc: 0.420000
(Iteration 501 / 2450) loss: 1.705528
(Iteration 601 / 2450) loss: 1.791422
(Iteration 701 / 2450) loss: 1.506501
(Epoch 3 / 10) train acc: 0.434000; val_acc: 0.450000
(Iteration 801 / 2450) loss: 1.618575
(Iteration 901 / 2450) loss: 1.685023
(Epoch 4 / 10) train acc: 0.464000; val_acc: 0.470000
(Iteration 1001 / 2450) loss: 1.571062
(Iteration 1101 / 2450) loss: 1.495980
(Iteration 1201 / 2450) loss: 1.517680
(Epoch 5 / 10) train acc: 0.462000; val_acc: 0.471000
(Iteration 1301 / 2450) loss: 1.474196
(Iteration 1401 / 2450) loss: 1.475637
(Epoch 6 / 10) train acc: 0.474000; val_acc: 0.465000
(Iteration 1501 / 2450) loss: 1.523059
(Iteration 1601 / 2450) loss: 1.591615
(Iteration 1701 / 2450) loss: 1.548040
(Epoch 7 / 10) train acc: 0.482000; val_acc: 0.485000
(Iteration 1801 / 2450) loss: 1.461553
(Iteration 1901 / 2450) loss: 1.426346
(Epoch 8 / 10) train acc: 0.495000; val_acc: 0.475000
(Iteration 2001 / 2450) loss: 1.385208
(Iteration 2101 / 2450) loss: 1.552263
```

```
(Iteration 2201 / 2450) loss: 1.345580
(Epoch 9 / 10) train acc: 0.525000; val_acc: 0.477000
(Iteration 2301 / 2450) loss: 1.393576
(Iteration 2401 / 2450) loss: 1.390657
(Epoch 10 / 10) train acc: 0.522000; val_acc: 0.482000
```

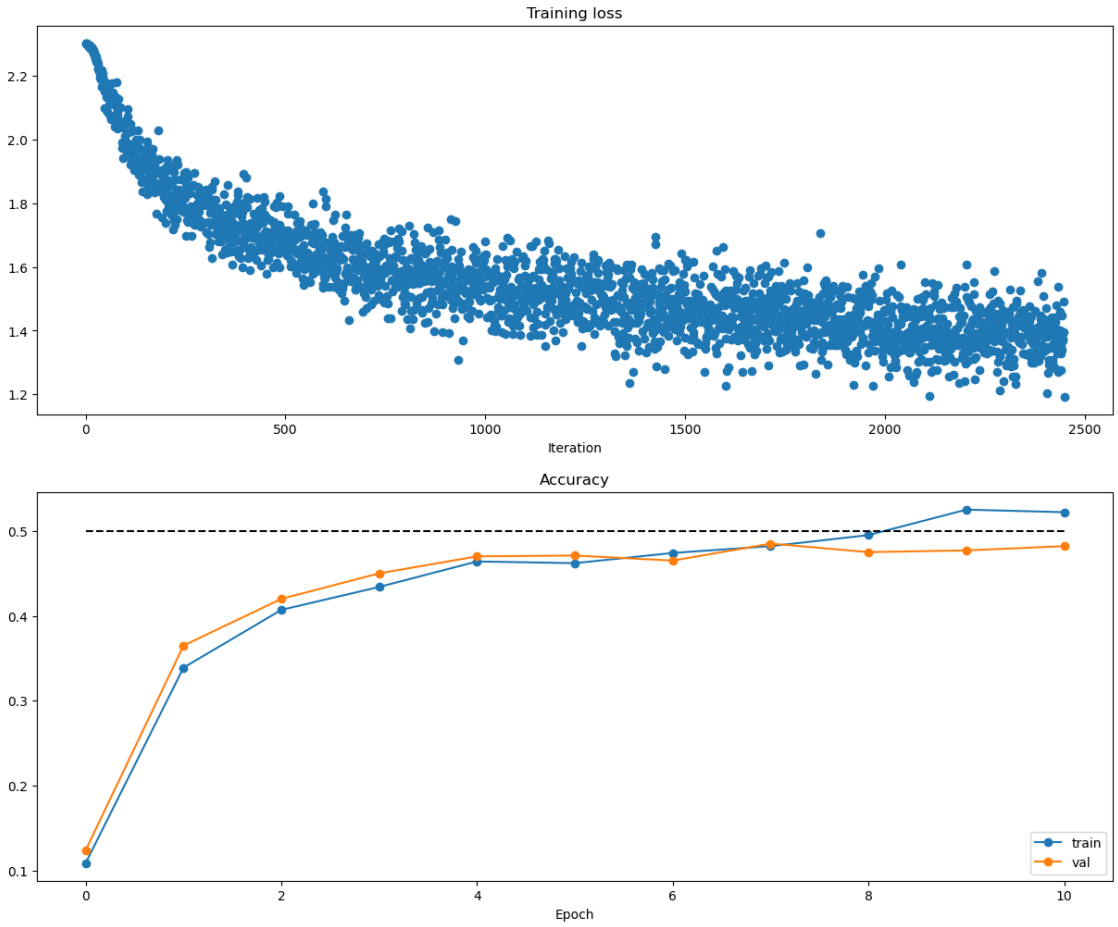
10 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

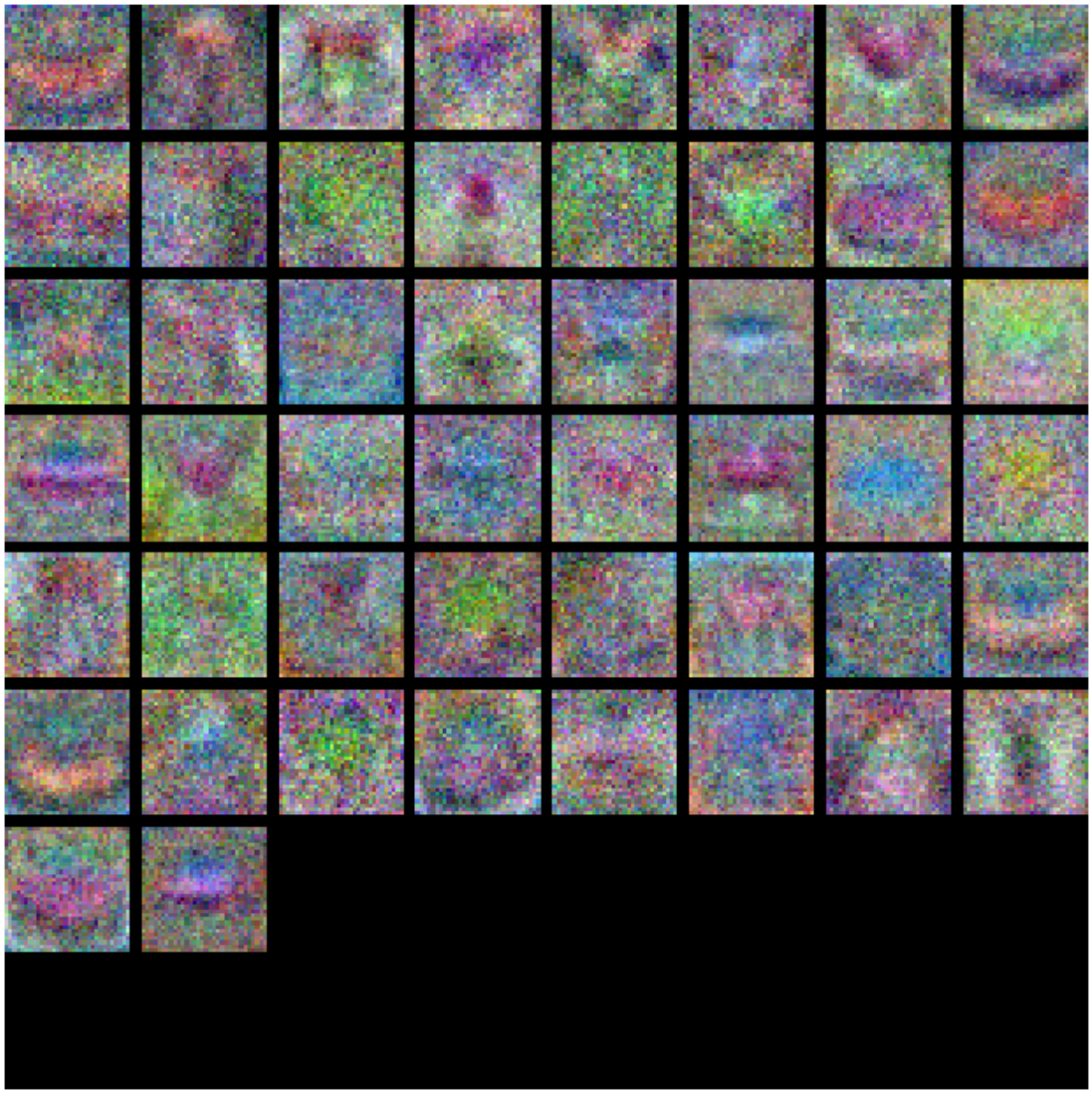
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [36]:
```
# Run this cell to visualize training loss and train / val accuracy

        plt.subplot(2, 1, 1)
        plt.title('Training loss')
        plt.plot(solver.loss_history, 'o')
        plt.xlabel('Iteration')

        plt.subplot(2, 1, 2)
        plt.title('Accuracy')
        plt.plot(solver.train_acc_history, '-o', label='train')
        plt.plot(solver.val_acc_history, '-o', label='val')
        plt.plot([0.5] * len(solver.val_acc_history), 'k--')
        plt.xlabel('Epoch')
        plt.legend(loc='lower right')
        plt.gcf().set_size_inches(15, 12)
        plt.show()
```



In [37]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

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

show_net_weights(model)
```



11 Tune your hyperparameters

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

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

also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

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

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [38]:
```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_model.                                                          #
#                                                                               #
# 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 thexs previous exercises.                          #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
best_model = model
pass

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

12 Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

In [39]:
```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.485
```

In [40]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.49
```

12.1 Inline Question 2:

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.

YourAnswer : 1,2,3
YourExplanation :

```
In [ ]:
```

features

September 30, 2023

```
In [ ]: # 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. 'cs231n/assignments/assignment1/'
        FOLDERNAME = None
        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/$FOLDERNAME/cs231n/datasets/
        !bash get_datasets.sh
        %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Image features exercise

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

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

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

```
In [1]: import random
        import numpy as np
        from cs231n.data_utils import load_CIFAR10
        import matplotlib.pyplot as plt
```

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

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

1.1 Load data

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

```
In [4]: from cs231n.features import color_histogram_hsv, hog_feature

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

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

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

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

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

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

1.2 Extract Features

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

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

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

```
In [5]: from cs231n.features import *

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

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

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

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
```

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

1.3 Train SVM on features

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

In [12]:
```
# Use the validation set to tune the learning learning_rate and regularization streng

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = np.logspace(-6, -2.7, 40)
regularization_strengths = np.logspace(0.07, 3, 10)

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning learning_rate and regularization strengt
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
results_array = np.zeros((regularization_strengths.shape[0], learning_rates.shape[0]))
for learning_rate in learning_rates:
  for reg in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train_feats, y_train, learning_rate, reg)
    train_acc = np.mean(svm.predict(X_train_feats) == y_train)
    pred = svm.predict(X_val_feats)
    cor = np.sum(pred == y_val)
    acc = float(cor) / y_val.shape[0]
    results[(learning_rate, reg)] = (train_acc, acc)
    results_array[np.where(regularization_strengths == reg)[0], np.where(learning_rate

    if acc > best_val:
      best_val = acc
      best_svm = svm
pass

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

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

print('best validation accuracy achieved: %f' % best_val)
```

```
lr 1.000000e-06 reg 1.174898e+00 train accuracy: 0.118714 val accuracy: 0.099000
```

```
lr 1.000000e-06 reg 2.486312e+00 train accuracy: 0.112857 val accuracy: 0.113000
lr 1.000000e-06 reg 5.261519e+00 train accuracy: 0.116143 val accuracy: 0.101000
lr 1.000000e-06 reg 1.113440e+01 train accuracy: 0.122204 val accuracy: 0.115000
lr 1.000000e-06 reg 2.356255e+01 train accuracy: 0.126286 val accuracy: 0.131000
lr 1.000000e-06 reg 4.986293e+01 train accuracy: 0.114204 val accuracy: 0.133000
lr 1.000000e-06 reg 1.055196e+02 train accuracy: 0.116163 val accuracy: 0.106000
lr 1.000000e-06 reg 2.233001e+02 train accuracy: 0.124714 val accuracy: 0.147000
lr 1.000000e-06 reg 4.725464e+02 train accuracy: 0.091898 val accuracy: 0.078000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.106571 val accuracy: 0.121000
lr 1.215109e-06 reg 1.174898e+00 train accuracy: 0.117469 val accuracy: 0.113000
lr 1.215109e-06 reg 2.486312e+00 train accuracy: 0.139939 val accuracy: 0.151000
lr 1.215109e-06 reg 5.261519e+00 train accuracy: 0.123673 val accuracy: 0.123000
lr 1.215109e-06 reg 1.113440e+01 train accuracy: 0.086612 val accuracy: 0.091000
lr 1.215109e-06 reg 2.356255e+01 train accuracy: 0.121653 val accuracy: 0.128000
lr 1.215109e-06 reg 4.986293e+01 train accuracy: 0.111000 val accuracy: 0.123000
lr 1.215109e-06 reg 1.055196e+02 train accuracy: 0.104510 val accuracy: 0.126000
lr 1.215109e-06 reg 2.233001e+02 train accuracy: 0.124878 val accuracy: 0.120000
lr 1.215109e-06 reg 4.725464e+02 train accuracy: 0.119020 val accuracy: 0.126000
lr 1.215109e-06 reg 1.000000e+03 train accuracy: 0.117694 val accuracy: 0.100000
lr 1.476491e-06 reg 1.174898e+00 train accuracy: 0.112857 val accuracy: 0.094000
lr 1.476491e-06 reg 2.486312e+00 train accuracy: 0.114429 val accuracy: 0.121000
lr 1.476491e-06 reg 5.261519e+00 train accuracy: 0.113531 val accuracy: 0.129000
lr 1.476491e-06 reg 1.113440e+01 train accuracy: 0.098551 val accuracy: 0.090000
lr 1.476491e-06 reg 2.356255e+01 train accuracy: 0.109245 val accuracy: 0.087000
lr 1.476491e-06 reg 4.986293e+01 train accuracy: 0.120816 val accuracy: 0.119000
lr 1.476491e-06 reg 1.055196e+02 train accuracy: 0.108755 val accuracy: 0.110000
lr 1.476491e-06 reg 2.233001e+02 train accuracy: 0.112510 val accuracy: 0.117000
lr 1.476491e-06 reg 4.725464e+02 train accuracy: 0.124735 val accuracy: 0.128000
lr 1.476491e-06 reg 1.000000e+03 train accuracy: 0.131306 val accuracy: 0.139000
lr 1.794098e-06 reg 1.174898e+00 train accuracy: 0.126857 val accuracy: 0.125000
lr 1.794098e-06 reg 2.486312e+00 train accuracy: 0.128735 val accuracy: 0.121000
lr 1.794098e-06 reg 5.261519e+00 train accuracy: 0.100490 val accuracy: 0.118000
lr 1.794098e-06 reg 1.113440e+01 train accuracy: 0.137327 val accuracy: 0.131000
lr 1.794098e-06 reg 2.356255e+01 train accuracy: 0.122469 val accuracy: 0.133000
lr 1.794098e-06 reg 4.986293e+01 train accuracy: 0.145224 val accuracy: 0.140000
lr 1.794098e-06 reg 1.055196e+02 train accuracy: 0.119776 val accuracy: 0.114000
lr 1.794098e-06 reg 2.233001e+02 train accuracy: 0.124286 val accuracy: 0.132000
lr 1.794098e-06 reg 4.725464e+02 train accuracy: 0.121020 val accuracy: 0.107000
lr 1.794098e-06 reg 1.000000e+03 train accuracy: 0.107143 val accuracy: 0.123000
lr 2.180025e-06 reg 1.174898e+00 train accuracy: 0.107694 val accuracy: 0.106000
lr 2.180025e-06 reg 2.486312e+00 train accuracy: 0.116102 val accuracy: 0.098000
lr 2.180025e-06 reg 5.261519e+00 train accuracy: 0.151714 val accuracy: 0.171000
lr 2.180025e-06 reg 1.113440e+01 train accuracy: 0.113816 val accuracy: 0.121000
lr 2.180025e-06 reg 2.356255e+01 train accuracy: 0.149694 val accuracy: 0.151000
lr 2.180025e-06 reg 4.986293e+01 train accuracy: 0.130571 val accuracy: 0.139000
lr 2.180025e-06 reg 1.055196e+02 train accuracy: 0.112531 val accuracy: 0.120000
lr 2.180025e-06 reg 2.233001e+02 train accuracy: 0.145449 val accuracy: 0.140000
lr 2.180025e-06 reg 4.725464e+02 train accuracy: 0.107816 val accuracy: 0.138000
```

```
lr 2.180025e-06 reg 1.000000e+03 train accuracy: 0.148102 val accuracy: 0.190000
lr 2.648969e-06 reg 1.174898e+00 train accuracy: 0.132102 val accuracy: 0.133000
lr 2.648969e-06 reg 2.486312e+00 train accuracy: 0.140776 val accuracy: 0.152000
lr 2.648969e-06 reg 5.261519e+00 train accuracy: 0.125245 val accuracy: 0.125000
lr 2.648969e-06 reg 1.113440e+01 train accuracy: 0.117939 val accuracy: 0.123000
lr 2.648969e-06 reg 2.356255e+01 train accuracy: 0.129653 val accuracy: 0.129000
lr 2.648969e-06 reg 4.986293e+01 train accuracy: 0.120633 val accuracy: 0.125000
lr 2.648969e-06 reg 1.055196e+02 train accuracy: 0.120245 val accuracy: 0.116000
lr 2.648969e-06 reg 2.233001e+02 train accuracy: 0.117878 val accuracy: 0.148000
lr 2.648969e-06 reg 4.725464e+02 train accuracy: 0.124429 val accuracy: 0.126000
lr 2.648969e-06 reg 1.000000e+03 train accuracy: 0.156143 val accuracy: 0.151000
lr 3.218788e-06 reg 1.174898e+00 train accuracy: 0.145306 val accuracy: 0.137000
lr 3.218788e-06 reg 2.486312e+00 train accuracy: 0.105531 val accuracy: 0.101000
lr 3.218788e-06 reg 5.261519e+00 train accuracy: 0.149265 val accuracy: 0.154000
lr 3.218788e-06 reg 1.113440e+01 train accuracy: 0.134347 val accuracy: 0.133000
lr 3.218788e-06 reg 2.356255e+01 train accuracy: 0.148041 val accuracy: 0.144000
lr 3.218788e-06 reg 4.986293e+01 train accuracy: 0.123612 val accuracy: 0.126000
lr 3.218788e-06 reg 1.055196e+02 train accuracy: 0.140082 val accuracy: 0.150000
lr 3.218788e-06 reg 2.233001e+02 train accuracy: 0.149184 val accuracy: 0.148000
lr 3.218788e-06 reg 4.725464e+02 train accuracy: 0.131429 val accuracy: 0.144000
lr 3.218788e-06 reg 1.000000e+03 train accuracy: 0.162531 val accuracy: 0.182000
lr 3.911179e-06 reg 1.174898e+00 train accuracy: 0.160367 val accuracy: 0.139000
lr 3.911179e-06 reg 2.486312e+00 train accuracy: 0.116939 val accuracy: 0.104000
lr 3.911179e-06 reg 5.261519e+00 train accuracy: 0.124020 val accuracy: 0.135000
lr 3.911179e-06 reg 1.113440e+01 train accuracy: 0.185245 val accuracy: 0.168000
lr 3.911179e-06 reg 2.356255e+01 train accuracy: 0.146633 val accuracy: 0.172000
lr 3.911179e-06 reg 4.986293e+01 train accuracy: 0.173143 val accuracy: 0.166000
lr 3.911179e-06 reg 1.055196e+02 train accuracy: 0.143898 val accuracy: 0.158000
lr 3.911179e-06 reg 2.233001e+02 train accuracy: 0.160265 val accuracy: 0.153000
lr 3.911179e-06 reg 4.725464e+02 train accuracy: 0.170633 val accuracy: 0.153000
lr 3.911179e-06 reg 1.000000e+03 train accuracy: 0.165653 val accuracy: 0.147000
lr 4.752510e-06 reg 1.174898e+00 train accuracy: 0.135776 val accuracy: 0.144000
lr 4.752510e-06 reg 2.486312e+00 train accuracy: 0.173184 val accuracy: 0.179000
lr 4.752510e-06 reg 5.261519e+00 train accuracy: 0.134367 val accuracy: 0.126000
lr 4.752510e-06 reg 1.113440e+01 train accuracy: 0.152061 val accuracy: 0.163000
lr 4.752510e-06 reg 2.356255e+01 train accuracy: 0.164306 val accuracy: 0.165000
lr 4.752510e-06 reg 4.986293e+01 train accuracy: 0.170041 val accuracy: 0.155000
lr 4.752510e-06 reg 1.055196e+02 train accuracy: 0.169041 val accuracy: 0.175000
lr 4.752510e-06 reg 2.233001e+02 train accuracy: 0.199286 val accuracy: 0.213000
lr 4.752510e-06 reg 4.725464e+02 train accuracy: 0.155143 val accuracy: 0.156000
lr 4.752510e-06 reg 1.000000e+03 train accuracy: 0.210449 val accuracy: 0.220000
lr 5.774820e-06 reg 1.174898e+00 train accuracy: 0.165122 val accuracy: 0.181000
lr 5.774820e-06 reg 2.486312e+00 train accuracy: 0.131551 val accuracy: 0.124000
lr 5.774820e-06 reg 5.261519e+00 train accuracy: 0.185959 val accuracy: 0.188000
lr 5.774820e-06 reg 1.113440e+01 train accuracy: 0.160592 val accuracy: 0.160000
lr 5.774820e-06 reg 2.356255e+01 train accuracy: 0.139653 val accuracy: 0.129000
lr 5.774820e-06 reg 4.986293e+01 train accuracy: 0.162898 val accuracy: 0.157000
lr 5.774820e-06 reg 1.055196e+02 train accuracy: 0.187367 val accuracy: 0.209000
```

```
lr 5.774820e-06 reg 2.233001e+02 train accuracy: 0.191163 val accuracy: 0.199000
lr 5.774820e-06 reg 4.725464e+02 train accuracy: 0.172163 val accuracy: 0.174000
lr 5.774820e-06 reg 1.000000e+03 train accuracy: 0.223082 val accuracy: 0.223000
lr 7.017038e-06 reg 1.174898e+00 train accuracy: 0.185122 val accuracy: 0.204000
lr 7.017038e-06 reg 2.486312e+00 train accuracy: 0.193224 val accuracy: 0.188000
lr 7.017038e-06 reg 5.261519e+00 train accuracy: 0.173306 val accuracy: 0.167000
lr 7.017038e-06 reg 1.113440e+01 train accuracy: 0.192102 val accuracy: 0.185000
lr 7.017038e-06 reg 2.356255e+01 train accuracy: 0.196469 val accuracy: 0.186000
lr 7.017038e-06 reg 4.986293e+01 train accuracy: 0.165143 val accuracy: 0.179000
lr 7.017038e-06 reg 1.055196e+02 train accuracy: 0.174816 val accuracy: 0.167000
lr 7.017038e-06 reg 2.233001e+02 train accuracy: 0.216245 val accuracy: 0.230000
lr 7.017038e-06 reg 4.725464e+02 train accuracy: 0.227184 val accuracy: 0.224000
lr 7.017038e-06 reg 1.000000e+03 train accuracy: 0.250000 val accuracy: 0.255000
lr 8.526469e-06 reg 1.174898e+00 train accuracy: 0.179082 val accuracy: 0.201000
lr 8.526469e-06 reg 2.486312e+00 train accuracy: 0.212918 val accuracy: 0.240000
lr 8.526469e-06 reg 5.261519e+00 train accuracy: 0.197265 val accuracy: 0.189000
lr 8.526469e-06 reg 1.113440e+01 train accuracy: 0.183510 val accuracy: 0.175000
lr 8.526469e-06 reg 2.356255e+01 train accuracy: 0.202857 val accuracy: 0.167000
lr 8.526469e-06 reg 4.986293e+01 train accuracy: 0.187531 val accuracy: 0.197000
lr 8.526469e-06 reg 1.055196e+02 train accuracy: 0.184837 val accuracy: 0.182000
lr 8.526469e-06 reg 2.233001e+02 train accuracy: 0.184163 val accuracy: 0.178000
lr 8.526469e-06 reg 4.725464e+02 train accuracy: 0.234837 val accuracy: 0.254000
lr 8.526469e-06 reg 1.000000e+03 train accuracy: 0.293918 val accuracy: 0.286000
lr 1.036059e-05 reg 1.174898e+00 train accuracy: 0.204959 val accuracy: 0.204000
lr 1.036059e-05 reg 2.486312e+00 train accuracy: 0.234224 val accuracy: 0.242000
lr 1.036059e-05 reg 5.261519e+00 train accuracy: 0.230653 val accuracy: 0.268000
lr 1.036059e-05 reg 1.113440e+01 train accuracy: 0.214163 val accuracy: 0.203000
lr 1.036059e-05 reg 2.356255e+01 train accuracy: 0.218898 val accuracy: 0.222000
lr 1.036059e-05 reg 4.986293e+01 train accuracy: 0.211673 val accuracy: 0.204000
lr 1.036059e-05 reg 1.055196e+02 train accuracy: 0.200020 val accuracy: 0.195000
lr 1.036059e-05 reg 2.233001e+02 train accuracy: 0.240347 val accuracy: 0.238000
lr 1.036059e-05 reg 4.725464e+02 train accuracy: 0.280939 val accuracy: 0.269000
lr 1.036059e-05 reg 1.000000e+03 train accuracy: 0.354796 val accuracy: 0.341000
lr 1.258925e-05 reg 1.174898e+00 train accuracy: 0.221735 val accuracy: 0.225000
lr 1.258925e-05 reg 2.486312e+00 train accuracy: 0.240980 val accuracy: 0.260000
lr 1.258925e-05 reg 5.261519e+00 train accuracy: 0.230204 val accuracy: 0.232000
lr 1.258925e-05 reg 1.113440e+01 train accuracy: 0.239776 val accuracy: 0.234000
lr 1.258925e-05 reg 2.356255e+01 train accuracy: 0.236735 val accuracy: 0.224000
lr 1.258925e-05 reg 4.986293e+01 train accuracy: 0.243673 val accuracy: 0.229000
lr 1.258925e-05 reg 1.055196e+02 train accuracy: 0.231796 val accuracy: 0.234000
lr 1.258925e-05 reg 2.233001e+02 train accuracy: 0.258163 val accuracy: 0.270000
lr 1.258925e-05 reg 4.725464e+02 train accuracy: 0.300163 val accuracy: 0.287000
lr 1.258925e-05 reg 1.000000e+03 train accuracy: 0.377735 val accuracy: 0.373000
lr 1.529732e-05 reg 1.174898e+00 train accuracy: 0.256673 val accuracy: 0.242000
lr 1.529732e-05 reg 2.486312e+00 train accuracy: 0.256490 val accuracy: 0.280000
lr 1.529732e-05 reg 5.261519e+00 train accuracy: 0.221163 val accuracy: 0.195000
lr 1.529732e-05 reg 1.113440e+01 train accuracy: 0.250531 val accuracy: 0.211000
lr 1.529732e-05 reg 2.356255e+01 train accuracy: 0.263694 val accuracy: 0.282000
```

```
lr 1.529732e-05 reg 4.986293e+01 train accuracy: 0.251163 val accuracy: 0.256000
lr 1.529732e-05 reg 1.055196e+02 train accuracy: 0.285408 val accuracy: 0.269000
lr 1.529732e-05 reg 2.233001e+02 train accuracy: 0.287224 val accuracy: 0.269000
lr 1.529732e-05 reg 4.725464e+02 train accuracy: 0.368102 val accuracy: 0.396000
lr 1.529732e-05 reg 1.000000e+03 train accuracy: 0.401102 val accuracy: 0.391000
lr 1.858792e-05 reg 1.174898e+00 train accuracy: 0.272490 val accuracy: 0.284000
lr 1.858792e-05 reg 2.486312e+00 train accuracy: 0.299245 val accuracy: 0.303000
lr 1.858792e-05 reg 5.261519e+00 train accuracy: 0.257082 val accuracy: 0.267000
lr 1.858792e-05 reg 1.113440e+01 train accuracy: 0.266755 val accuracy: 0.284000
lr 1.858792e-05 reg 2.356255e+01 train accuracy: 0.271918 val accuracy: 0.283000
lr 1.858792e-05 reg 4.986293e+01 train accuracy: 0.283673 val accuracy: 0.278000
lr 1.858792e-05 reg 1.055196e+02 train accuracy: 0.299408 val accuracy: 0.296000
lr 1.858792e-05 reg 2.233001e+02 train accuracy: 0.311041 val accuracy: 0.313000
lr 1.858792e-05 reg 4.725464e+02 train accuracy: 0.368429 val accuracy: 0.348000
lr 1.858792e-05 reg 1.000000e+03 train accuracy: 0.412286 val accuracy: 0.413000
lr 2.258636e-05 reg 1.174898e+00 train accuracy: 0.278653 val accuracy: 0.283000
lr 2.258636e-05 reg 2.486312e+00 train accuracy: 0.306653 val accuracy: 0.309000
lr 2.258636e-05 reg 5.261519e+00 train accuracy: 0.286959 val accuracy: 0.285000
lr 2.258636e-05 reg 1.113440e+01 train accuracy: 0.299898 val accuracy: 0.317000
lr 2.258636e-05 reg 2.356255e+01 train accuracy: 0.305327 val accuracy: 0.310000
lr 2.258636e-05 reg 4.986293e+01 train accuracy: 0.298673 val accuracy: 0.311000
lr 2.258636e-05 reg 1.055196e+02 train accuracy: 0.332837 val accuracy: 0.319000
lr 2.258636e-05 reg 2.233001e+02 train accuracy: 0.354551 val accuracy: 0.360000
lr 2.258636e-05 reg 4.725464e+02 train accuracy: 0.387122 val accuracy: 0.381000
lr 2.258636e-05 reg 1.000000e+03 train accuracy: 0.412551 val accuracy: 0.415000
lr 2.744489e-05 reg 1.174898e+00 train accuracy: 0.321122 val accuracy: 0.327000
lr 2.744489e-05 reg 2.486312e+00 train accuracy: 0.305816 val accuracy: 0.313000
lr 2.744489e-05 reg 5.261519e+00 train accuracy: 0.320082 val accuracy: 0.328000
lr 2.744489e-05 reg 1.113440e+01 train accuracy: 0.323245 val accuracy: 0.324000
lr 2.744489e-05 reg 2.356255e+01 train accuracy: 0.318102 val accuracy: 0.298000
lr 2.744489e-05 reg 4.986293e+01 train accuracy: 0.342082 val accuracy: 0.351000
lr 2.744489e-05 reg 1.055196e+02 train accuracy: 0.339122 val accuracy: 0.342000
lr 2.744489e-05 reg 2.233001e+02 train accuracy: 0.372551 val accuracy: 0.376000
lr 2.744489e-05 reg 4.725464e+02 train accuracy: 0.403694 val accuracy: 0.410000
lr 2.744489e-05 reg 1.000000e+03 train accuracy: 0.405204 val accuracy: 0.402000
lr 3.334855e-05 reg 1.174898e+00 train accuracy: 0.342980 val accuracy: 0.324000
lr 3.334855e-05 reg 2.486312e+00 train accuracy: 0.345367 val accuracy: 0.364000
lr 3.334855e-05 reg 5.261519e+00 train accuracy: 0.347776 val accuracy: 0.357000
lr 3.334855e-05 reg 1.113440e+01 train accuracy: 0.337551 val accuracy: 0.353000
lr 3.334855e-05 reg 2.356255e+01 train accuracy: 0.350245 val accuracy: 0.373000
lr 3.334855e-05 reg 4.986293e+01 train accuracy: 0.367184 val accuracy: 0.350000
lr 3.334855e-05 reg 1.055196e+02 train accuracy: 0.357388 val accuracy: 0.357000
lr 3.334855e-05 reg 2.233001e+02 train accuracy: 0.375163 val accuracy: 0.370000
lr 3.334855e-05 reg 4.725464e+02 train accuracy: 0.410918 val accuracy: 0.409000
lr 3.334855e-05 reg 1.000000e+03 train accuracy: 0.409612 val accuracy: 0.411000
lr 4.052213e-05 reg 1.174898e+00 train accuracy: 0.364612 val accuracy: 0.367000
lr 4.052213e-05 reg 2.486312e+00 train accuracy: 0.365939 val accuracy: 0.381000
lr 4.052213e-05 reg 5.261519e+00 train accuracy: 0.354306 val accuracy: 0.357000
```

```
lr 4.052213e-05 reg 1.113440e+01 train accuracy: 0.358755 val accuracy: 0.358000
lr 4.052213e-05 reg 2.356255e+01 train accuracy: 0.364776 val accuracy: 0.357000
lr 4.052213e-05 reg 4.986293e+01 train accuracy: 0.349510 val accuracy: 0.354000
lr 4.052213e-05 reg 1.055196e+02 train accuracy: 0.382837 val accuracy: 0.396000
lr 4.052213e-05 reg 2.233001e+02 train accuracy: 0.403755 val accuracy: 0.410000
lr 4.052213e-05 reg 4.725464e+02 train accuracy: 0.410735 val accuracy: 0.408000
lr 4.052213e-05 reg 1.000000e+03 train accuracy: 0.411408 val accuracy: 0.417000
lr 4.923883e-05 reg 1.174898e+00 train accuracy: 0.374224 val accuracy: 0.377000
lr 4.923883e-05 reg 2.486312e+00 train accuracy: 0.364265 val accuracy: 0.356000
lr 4.923883e-05 reg 5.261519e+00 train accuracy: 0.369041 val accuracy: 0.364000
lr 4.923883e-05 reg 1.113440e+01 train accuracy: 0.375082 val accuracy: 0.365000
lr 4.923883e-05 reg 2.356255e+01 train accuracy: 0.364327 val accuracy: 0.360000
lr 4.923883e-05 reg 4.986293e+01 train accuracy: 0.393653 val accuracy: 0.403000
lr 4.923883e-05 reg 1.055196e+02 train accuracy: 0.399939 val accuracy: 0.391000
lr 4.923883e-05 reg 2.233001e+02 train accuracy: 0.413816 val accuracy: 0.417000
lr 4.923883e-05 reg 4.725464e+02 train accuracy: 0.408898 val accuracy: 0.406000
lr 4.923883e-05 reg 1.000000e+03 train accuracy: 0.411306 val accuracy: 0.407000
lr 5.983056e-05 reg 1.174898e+00 train accuracy: 0.383265 val accuracy: 0.383000
lr 5.983056e-05 reg 2.486312e+00 train accuracy: 0.372449 val accuracy: 0.371000
lr 5.983056e-05 reg 5.261519e+00 train accuracy: 0.384122 val accuracy: 0.396000
lr 5.983056e-05 reg 1.113440e+01 train accuracy: 0.381918 val accuracy: 0.390000
lr 5.983056e-05 reg 2.356255e+01 train accuracy: 0.388612 val accuracy: 0.377000
lr 5.983056e-05 reg 4.986293e+01 train accuracy: 0.394592 val accuracy: 0.386000
lr 5.983056e-05 reg 1.055196e+02 train accuracy: 0.409776 val accuracy: 0.416000
lr 5.983056e-05 reg 2.233001e+02 train accuracy: 0.409265 val accuracy: 0.410000
lr 5.983056e-05 reg 4.725464e+02 train accuracy: 0.407245 val accuracy: 0.419000
lr 5.983056e-05 reg 1.000000e+03 train accuracy: 0.402714 val accuracy: 0.394000
lr 7.270068e-05 reg 1.174898e+00 train accuracy: 0.388102 val accuracy: 0.397000
lr 7.270068e-05 reg 2.486312e+00 train accuracy: 0.388245 val accuracy: 0.377000
lr 7.270068e-05 reg 5.261519e+00 train accuracy: 0.389347 val accuracy: 0.396000
lr 7.270068e-05 reg 1.113440e+01 train accuracy: 0.395449 val accuracy: 0.413000
lr 7.270068e-05 reg 2.356255e+01 train accuracy: 0.394571 val accuracy: 0.388000
lr 7.270068e-05 reg 4.986293e+01 train accuracy: 0.399204 val accuracy: 0.385000
lr 7.270068e-05 reg 1.055196e+02 train accuracy: 0.411918 val accuracy: 0.425000
lr 7.270068e-05 reg 2.233001e+02 train accuracy: 0.415163 val accuracy: 0.417000
lr 7.270068e-05 reg 4.725464e+02 train accuracy: 0.402816 val accuracy: 0.401000
lr 7.270068e-05 reg 1.000000e+03 train accuracy: 0.390408 val accuracy: 0.387000
lr 8.833928e-05 reg 1.174898e+00 train accuracy: 0.399959 val accuracy: 0.405000
lr 8.833928e-05 reg 2.486312e+00 train accuracy: 0.400551 val accuracy: 0.383000
lr 8.833928e-05 reg 5.261519e+00 train accuracy: 0.388816 val accuracy: 0.388000
lr 8.833928e-05 reg 1.113440e+01 train accuracy: 0.397347 val accuracy: 0.391000
lr 8.833928e-05 reg 2.356255e+01 train accuracy: 0.400612 val accuracy: 0.406000
lr 8.833928e-05 reg 4.986293e+01 train accuracy: 0.412408 val accuracy: 0.413000
lr 8.833928e-05 reg 1.055196e+02 train accuracy: 0.408980 val accuracy: 0.401000
lr 8.833928e-05 reg 2.233001e+02 train accuracy: 0.405367 val accuracy: 0.397000
lr 8.833928e-05 reg 4.725464e+02 train accuracy: 0.406592 val accuracy: 0.405000
lr 8.833928e-05 reg 1.000000e+03 train accuracy: 0.394796 val accuracy: 0.379000
lr 1.073419e-04 reg 1.174898e+00 train accuracy: 0.406347 val accuracy: 0.407000
```

```
lr 1.073419e-04 reg 2.486312e+00 train accuracy: 0.409265 val accuracy: 0.404000
lr 1.073419e-04 reg 5.261519e+00 train accuracy: 0.405041 val accuracy: 0.401000
lr 1.073419e-04 reg 1.113440e+01 train accuracy: 0.406082 val accuracy: 0.397000
lr 1.073419e-04 reg 2.356255e+01 train accuracy: 0.401857 val accuracy: 0.401000
lr 1.073419e-04 reg 4.986293e+01 train accuracy: 0.408122 val accuracy: 0.398000
lr 1.073419e-04 reg 1.055196e+02 train accuracy: 0.416163 val accuracy: 0.418000
lr 1.073419e-04 reg 2.233001e+02 train accuracy: 0.410673 val accuracy: 0.408000
lr 1.073419e-04 reg 4.725464e+02 train accuracy: 0.407755 val accuracy: 0.392000
lr 1.073419e-04 reg 1.000000e+03 train accuracy: 0.391816 val accuracy: 0.385000
lr 1.304321e-04 reg 1.174898e+00 train accuracy: 0.407898 val accuracy: 0.412000
lr 1.304321e-04 reg 2.486312e+00 train accuracy: 0.401939 val accuracy: 0.394000
lr 1.304321e-04 reg 5.261519e+00 train accuracy: 0.409388 val accuracy: 0.423000
lr 1.304321e-04 reg 1.113440e+01 train accuracy: 0.407143 val accuracy: 0.419000
lr 1.304321e-04 reg 2.356255e+01 train accuracy: 0.414857 val accuracy: 0.407000
lr 1.304321e-04 reg 4.986293e+01 train accuracy: 0.405286 val accuracy: 0.383000
lr 1.304321e-04 reg 1.055196e+02 train accuracy: 0.416020 val accuracy: 0.419000
lr 1.304321e-04 reg 2.233001e+02 train accuracy: 0.405939 val accuracy: 0.407000
lr 1.304321e-04 reg 4.725464e+02 train accuracy: 0.408388 val accuracy: 0.403000
lr 1.304321e-04 reg 1.000000e+03 train accuracy: 0.396980 val accuracy: 0.383000
lr 1.584893e-04 reg 1.174898e+00 train accuracy: 0.403122 val accuracy: 0.391000
lr 1.584893e-04 reg 2.486312e+00 train accuracy: 0.408327 val accuracy: 0.410000
lr 1.584893e-04 reg 5.261519e+00 train accuracy: 0.410347 val accuracy: 0.413000
lr 1.584893e-04 reg 1.113440e+01 train accuracy: 0.406633 val accuracy: 0.396000
lr 1.584893e-04 reg 2.356255e+01 train accuracy: 0.412265 val accuracy: 0.416000
lr 1.584893e-04 reg 4.986293e+01 train accuracy: 0.411918 val accuracy: 0.425000
lr 1.584893e-04 reg 1.055196e+02 train accuracy: 0.412469 val accuracy: 0.401000
lr 1.584893e-04 reg 2.233001e+02 train accuracy: 0.409959 val accuracy: 0.431000
lr 1.584893e-04 reg 4.725464e+02 train accuracy: 0.397551 val accuracy: 0.413000
lr 1.584893e-04 reg 1.000000e+03 train accuracy: 0.385878 val accuracy: 0.375000
lr 1.925819e-04 reg 1.174898e+00 train accuracy: 0.408510 val accuracy: 0.410000
lr 1.925819e-04 reg 2.486312e+00 train accuracy: 0.413143 val accuracy: 0.423000
lr 1.925819e-04 reg 5.261519e+00 train accuracy: 0.406571 val accuracy: 0.406000
lr 1.925819e-04 reg 1.113440e+01 train accuracy: 0.412551 val accuracy: 0.416000
lr 1.925819e-04 reg 2.356255e+01 train accuracy: 0.416510 val accuracy: 0.418000
lr 1.925819e-04 reg 4.986293e+01 train accuracy: 0.415245 val accuracy: 0.418000
lr 1.925819e-04 reg 1.055196e+02 train accuracy: 0.403571 val accuracy: 0.397000
lr 1.925819e-04 reg 2.233001e+02 train accuracy: 0.403286 val accuracy: 0.423000
lr 1.925819e-04 reg 4.725464e+02 train accuracy: 0.405388 val accuracy: 0.400000
lr 1.925819e-04 reg 1.000000e+03 train accuracy: 0.376694 val accuracy: 0.409000
lr 2.340080e-04 reg 1.174898e+00 train accuracy: 0.412347 val accuracy: 0.422000
lr 2.340080e-04 reg 2.486312e+00 train accuracy: 0.410490 val accuracy: 0.403000
lr 2.340080e-04 reg 5.261519e+00 train accuracy: 0.416551 val accuracy: 0.425000
lr 2.340080e-04 reg 1.113440e+01 train accuracy: 0.413429 val accuracy: 0.410000
lr 2.340080e-04 reg 2.356255e+01 train accuracy: 0.412041 val accuracy: 0.412000
lr 2.340080e-04 reg 4.986293e+01 train accuracy: 0.412939 val accuracy: 0.414000
lr 2.340080e-04 reg 1.055196e+02 train accuracy: 0.406939 val accuracy: 0.402000
lr 2.340080e-04 reg 2.233001e+02 train accuracy: 0.410163 val accuracy: 0.404000
lr 2.340080e-04 reg 4.725464e+02 train accuracy: 0.403837 val accuracy: 0.402000
```

```
lr 2.340080e-04 reg 1.000000e+03 train accuracy: 0.386959 val accuracy: 0.395000
lr 2.843454e-04 reg 1.174898e+00 train accuracy: 0.416082 val accuracy: 0.427000
lr 2.843454e-04 reg 2.486312e+00 train accuracy: 0.420122 val accuracy: 0.422000
lr 2.843454e-04 reg 5.261519e+00 train accuracy: 0.410857 val accuracy: 0.409000
lr 2.843454e-04 reg 1.113440e+01 train accuracy: 0.411714 val accuracy: 0.400000
lr 2.843454e-04 reg 2.356255e+01 train accuracy: 0.413265 val accuracy: 0.415000
lr 2.843454e-04 reg 4.986293e+01 train accuracy: 0.420714 val accuracy: 0.429000
lr 2.843454e-04 reg 1.055196e+02 train accuracy: 0.405735 val accuracy: 0.403000
lr 2.843454e-04 reg 2.233001e+02 train accuracy: 0.401959 val accuracy: 0.392000
lr 2.843454e-04 reg 4.725464e+02 train accuracy: 0.381306 val accuracy: 0.391000
lr 2.843454e-04 reg 1.000000e+03 train accuracy: 0.365265 val accuracy: 0.366000
lr 3.455107e-04 reg 1.174898e+00 train accuracy: 0.419714 val accuracy: 0.427000
lr 3.455107e-04 reg 2.486312e+00 train accuracy: 0.412959 val accuracy: 0.415000
lr 3.455107e-04 reg 5.261519e+00 train accuracy: 0.411755 val accuracy: 0.410000
lr 3.455107e-04 reg 1.113440e+01 train accuracy: 0.414327 val accuracy: 0.422000
lr 3.455107e-04 reg 2.356255e+01 train accuracy: 0.416551 val accuracy: 0.430000
lr 3.455107e-04 reg 4.986293e+01 train accuracy: 0.418102 val accuracy: 0.420000
lr 3.455107e-04 reg 1.055196e+02 train accuracy: 0.410490 val accuracy: 0.408000
lr 3.455107e-04 reg 2.233001e+02 train accuracy: 0.408000 val accuracy: 0.413000
lr 3.455107e-04 reg 4.725464e+02 train accuracy: 0.392408 val accuracy: 0.385000
lr 3.455107e-04 reg 1.000000e+03 train accuracy: 0.371367 val accuracy: 0.364000
lr 4.198333e-04 reg 1.174898e+00 train accuracy: 0.415469 val accuracy: 0.420000
lr 4.198333e-04 reg 2.486312e+00 train accuracy: 0.413510 val accuracy: 0.418000
lr 4.198333e-04 reg 5.261519e+00 train accuracy: 0.417388 val accuracy: 0.423000
lr 4.198333e-04 reg 1.113440e+01 train accuracy: 0.415714 val accuracy: 0.414000
lr 4.198333e-04 reg 2.356255e+01 train accuracy: 0.417531 val accuracy: 0.422000
lr 4.198333e-04 reg 4.986293e+01 train accuracy: 0.412653 val accuracy: 0.420000
lr 4.198333e-04 reg 1.055196e+02 train accuracy: 0.410469 val accuracy: 0.416000
lr 4.198333e-04 reg 2.233001e+02 train accuracy: 0.405224 val accuracy: 0.420000
lr 4.198333e-04 reg 4.725464e+02 train accuracy: 0.378265 val accuracy: 0.381000
lr 4.198333e-04 reg 1.000000e+03 train accuracy: 0.322776 val accuracy: 0.307000
lr 5.101434e-04 reg 1.174898e+00 train accuracy: 0.416796 val accuracy: 0.413000
lr 5.101434e-04 reg 2.486312e+00 train accuracy: 0.417816 val accuracy: 0.421000
lr 5.101434e-04 reg 5.261519e+00 train accuracy: 0.416755 val accuracy: 0.413000
lr 5.101434e-04 reg 1.113440e+01 train accuracy: 0.414469 val accuracy: 0.426000
lr 5.101434e-04 reg 2.356255e+01 train accuracy: 0.411122 val accuracy: 0.412000
lr 5.101434e-04 reg 4.986293e+01 train accuracy: 0.413510 val accuracy: 0.410000
lr 5.101434e-04 reg 1.055196e+02 train accuracy: 0.396633 val accuracy: 0.394000
lr 5.101434e-04 reg 2.233001e+02 train accuracy: 0.391306 val accuracy: 0.382000
lr 5.101434e-04 reg 4.725464e+02 train accuracy: 0.372694 val accuracy: 0.384000
lr 5.101434e-04 reg 1.000000e+03 train accuracy: 0.329429 val accuracy: 0.323000
lr 6.198801e-04 reg 1.174898e+00 train accuracy: 0.424020 val accuracy: 0.420000
lr 6.198801e-04 reg 2.486312e+00 train accuracy: 0.419408 val accuracy: 0.410000
lr 6.198801e-04 reg 5.261519e+00 train accuracy: 0.418204 val accuracy: 0.426000
lr 6.198801e-04 reg 1.113440e+01 train accuracy: 0.416061 val accuracy: 0.425000
lr 6.198801e-04 reg 2.356255e+01 train accuracy: 0.418449 val accuracy: 0.427000
lr 6.198801e-04 reg 4.986293e+01 train accuracy: 0.409061 val accuracy: 0.404000
lr 6.198801e-04 reg 1.055196e+02 train accuracy: 0.410020 val accuracy: 0.410000
```

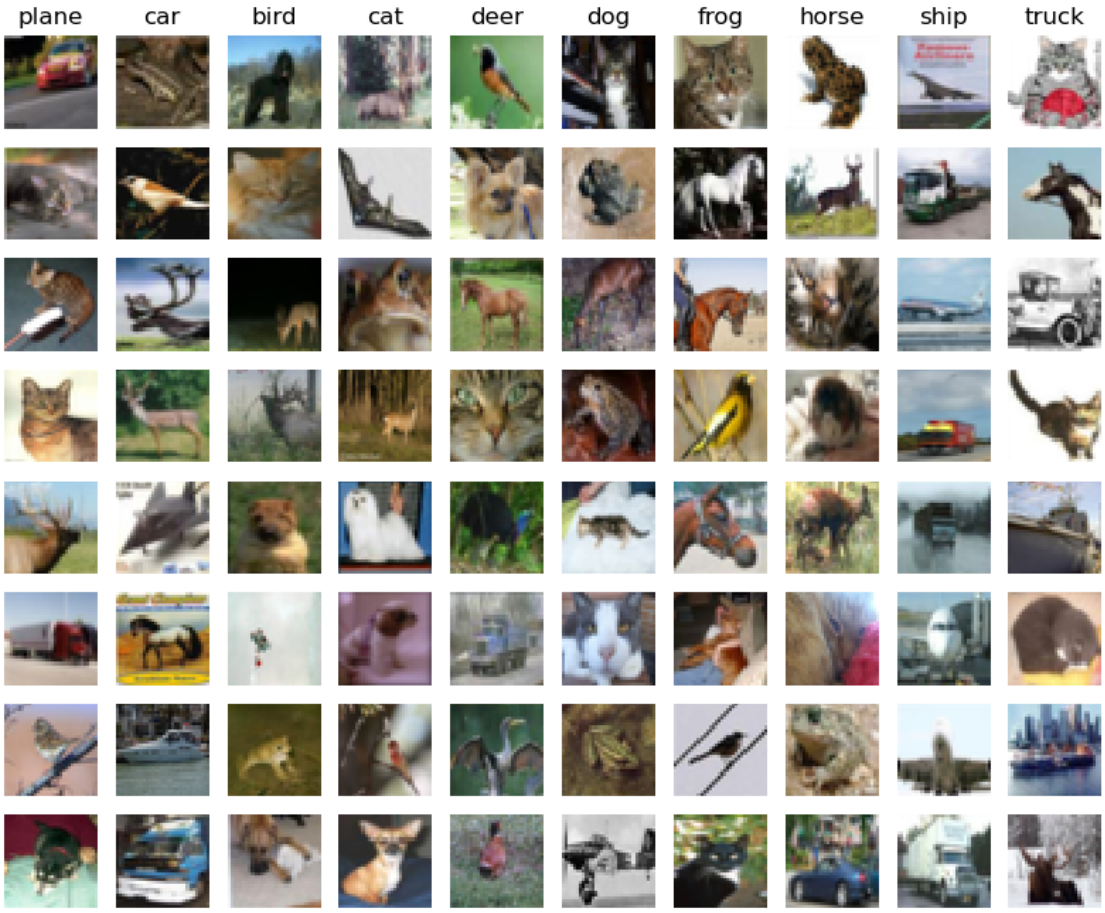
```
lr 6.198801e-04 reg 2.233001e+02 train accuracy: 0.382673 val accuracy: 0.392000
lr 6.198801e-04 reg 4.725464e+02 train accuracy: 0.349571 val accuracy: 0.327000
lr 6.198801e-04 reg 1.000000e+03 train accuracy: 0.294878 val accuracy: 0.281000
lr 7.532221e-04 reg 1.174898e+00 train accuracy: 0.430571 val accuracy: 0.433000
lr 7.532221e-04 reg 2.486312e+00 train accuracy: 0.427286 val accuracy: 0.439000
lr 7.532221e-04 reg 5.261519e+00 train accuracy: 0.422102 val accuracy: 0.422000
lr 7.532221e-04 reg 1.113440e+01 train accuracy: 0.417878 val accuracy: 0.419000
lr 7.532221e-04 reg 2.356255e+01 train accuracy: 0.407673 val accuracy: 0.401000
lr 7.532221e-04 reg 4.986293e+01 train accuracy: 0.410163 val accuracy: 0.402000
lr 7.532221e-04 reg 1.055196e+02 train accuracy: 0.386551 val accuracy: 0.364000
lr 7.532221e-04 reg 2.233001e+02 train accuracy: 0.379061 val accuracy: 0.361000
lr 7.532221e-04 reg 4.725464e+02 train accuracy: 0.330592 val accuracy: 0.321000
lr 7.532221e-04 reg 1.000000e+03 train accuracy: 0.292612 val accuracy: 0.305000
lr 9.152473e-04 reg 1.174898e+00 train accuracy: 0.437143 val accuracy: 0.444000
lr 9.152473e-04 reg 2.486312e+00 train accuracy: 0.429837 val accuracy: 0.427000
lr 9.152473e-04 reg 5.261519e+00 train accuracy: 0.433449 val accuracy: 0.437000
lr 9.152473e-04 reg 1.113440e+01 train accuracy: 0.422796 val accuracy: 0.423000
lr 9.152473e-04 reg 2.356255e+01 train accuracy: 0.410122 val accuracy: 0.421000
lr 9.152473e-04 reg 4.986293e+01 train accuracy: 0.412020 val accuracy: 0.414000
lr 9.152473e-04 reg 1.055196e+02 train accuracy: 0.396918 val accuracy: 0.389000
lr 9.152473e-04 reg 2.233001e+02 train accuracy: 0.388041 val accuracy: 0.381000
lr 9.152473e-04 reg 4.725464e+02 train accuracy: 0.345755 val accuracy: 0.359000
lr 9.152473e-04 reg 1.000000e+03 train accuracy: 0.228939 val accuracy: 0.213000
lr 1.112126e-03 reg 1.174898e+00 train accuracy: 0.438265 val accuracy: 0.438000
lr 1.112126e-03 reg 2.486312e+00 train accuracy: 0.436735 val accuracy: 0.430000
lr 1.112126e-03 reg 5.261519e+00 train accuracy: 0.433020 val accuracy: 0.428000
lr 1.112126e-03 reg 1.113440e+01 train accuracy: 0.420959 val accuracy: 0.423000
lr 1.112126e-03 reg 2.356255e+01 train accuracy: 0.410918 val accuracy: 0.416000
lr 1.112126e-03 reg 4.986293e+01 train accuracy: 0.409490 val accuracy: 0.411000
lr 1.112126e-03 reg 1.055196e+02 train accuracy: 0.393612 val accuracy: 0.393000
lr 1.112126e-03 reg 2.233001e+02 train accuracy: 0.360918 val accuracy: 0.343000
lr 1.112126e-03 reg 4.725464e+02 train accuracy: 0.304204 val accuracy: 0.300000
lr 1.112126e-03 reg 1.000000e+03 train accuracy: 0.081633 val accuracy: 0.073000
lr 1.351354e-03 reg 1.174898e+00 train accuracy: 0.446286 val accuracy: 0.450000
lr 1.351354e-03 reg 2.486312e+00 train accuracy: 0.443265 val accuracy: 0.437000
lr 1.351354e-03 reg 5.261519e+00 train accuracy: 0.437041 val accuracy: 0.441000
lr 1.351354e-03 reg 1.113440e+01 train accuracy: 0.422837 val accuracy: 0.423000
lr 1.351354e-03 reg 2.356255e+01 train accuracy: 0.401939 val accuracy: 0.389000
lr 1.351354e-03 reg 4.986293e+01 train accuracy: 0.398776 val accuracy: 0.409000
lr 1.351354e-03 reg 1.055196e+02 train accuracy: 0.390347 val accuracy: 0.381000
lr 1.351354e-03 reg 2.233001e+02 train accuracy: 0.334265 val accuracy: 0.343000
lr 1.351354e-03 reg 4.725464e+02 train accuracy: 0.301694 val accuracy: 0.290000
lr 1.351354e-03 reg 1.000000e+03 train accuracy: 0.076959 val accuracy: 0.075000
lr 1.642043e-03 reg 1.174898e+00 train accuracy: 0.449469 val accuracy: 0.447000
lr 1.642043e-03 reg 2.486312e+00 train accuracy: 0.445857 val accuracy: 0.445000
lr 1.642043e-03 reg 5.261519e+00 train accuracy: 0.437878 val accuracy: 0.441000
lr 1.642043e-03 reg 1.113440e+01 train accuracy: 0.421694 val accuracy: 0.425000
lr 1.642043e-03 reg 2.356255e+01 train accuracy: 0.413020 val accuracy: 0.408000
```

```
lr 1.642043e-03 reg 4.986293e+01 train accuracy: 0.375082 val accuracy: 0.373000
lr 1.642043e-03 reg 1.055196e+02 train accuracy: 0.376184 val accuracy: 0.364000
lr 1.642043e-03 reg 2.233001e+02 train accuracy: 0.349469 val accuracy: 0.333000
lr 1.642043e-03 reg 4.725464e+02 train accuracy: 0.266020 val accuracy: 0.300000
lr 1.642043e-03 reg 1.000000e+03 train accuracy: 0.066224 val accuracy: 0.049000
lr 1.995262e-03 reg 1.174898e+00 train accuracy: 0.455000 val accuracy: 0.451000
lr 1.995262e-03 reg 2.486312e+00 train accuracy: 0.451939 val accuracy: 0.446000
lr 1.995262e-03 reg 5.261519e+00 train accuracy: 0.442367 val accuracy: 0.452000
lr 1.995262e-03 reg 1.113440e+01 train accuracy: 0.430367 val accuracy: 0.430000
lr 1.995262e-03 reg 2.356255e+01 train accuracy: 0.398612 val accuracy: 0.399000
lr 1.995262e-03 reg 4.986293e+01 train accuracy: 0.396000 val accuracy: 0.399000
lr 1.995262e-03 reg 1.055196e+02 train accuracy: 0.372347 val accuracy: 0.349000
lr 1.995262e-03 reg 2.233001e+02 train accuracy: 0.330653 val accuracy: 0.327000
lr 1.995262e-03 reg 4.725464e+02 train accuracy: 0.178837 val accuracy: 0.194000
lr 1.995262e-03 reg 1.000000e+03 train accuracy: 0.051959 val accuracy: 0.050000
best validation accuracy achieved: 0.452000
```

In [13]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least 0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.46
```

In [14]:
```
# 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', 'tru
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls + 1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```



1.3.1 Inline question 1:

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

Your Answer : The misclassification results make sense. For example, the misclassified cat is classified as a dog. This is because the cat and the dog have similar features. The backgroud could also be a factor

1.4 Neural Network on image features

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

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

In [15]:
```
# 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 [31]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
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.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
learning_rate_decays = [0.99, 0.98, 0.7, 0.5]
learning_rates = np.logspace(-1, 0.5, 20)
regularization_strengths = np.logspace(-5, -7, 3)
best_val = -1
for rate in learning_rates:
  for dc in learning_rate_decays:
    solver = Solver(net, data, update_rule="sgd",
                    optim_config={"learning_rate" : rate * 10,}, lr_decay= dc,
                    num_epochs=5, batch_size=200, print_every=100)
    solver.train()
    if solver.best_val_acc > best_val:
      best_val = solver.best_val_acc
      best_net = net
```

```
            if solver.loss_history[-1] <0.5:
              break

        pass

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

```
(Iteration 1 / 1225) loss: 2.302583
(Epoch 0 / 5) train acc: 0.094000; val_acc: 0.079000
(Iteration 101 / 1225) loss: 1.366523
(Iteration 201 / 1225) loss: 1.201156
(Epoch 1 / 5) train acc: 0.540000; val_acc: 0.503000
(Iteration 301 / 1225) loss: 1.259281
(Iteration 401 / 1225) loss: 1.115117
(Epoch 2 / 5) train acc: 0.533000; val_acc: 0.507000
(Iteration 501 / 1225) loss: 1.406146
(Iteration 601 / 1225) loss: 0.997879
(Iteration 701 / 1225) loss: 1.006408
(Epoch 3 / 5) train acc: 0.608000; val_acc: 0.552000
(Iteration 801 / 1225) loss: 1.145008
(Iteration 901 / 1225) loss: 0.986440
(Epoch 4 / 5) train acc: 0.652000; val_acc: 0.550000
(Iteration 1001 / 1225) loss: 1.163182
(Iteration 1101 / 1225) loss: 0.845756
(Iteration 1201 / 1225) loss: 1.032019
(Epoch 5 / 5) train acc: 0.672000; val_acc: 0.559000
(Iteration 1 / 1225) loss: 1.140899
(Epoch 0 / 5) train acc: 0.696000; val_acc: 0.569000
(Iteration 101 / 1225) loss: 1.085187
(Iteration 201 / 1225) loss: 0.894590
(Epoch 1 / 5) train acc: 0.697000; val_acc: 0.553000
(Iteration 301 / 1225) loss: 0.750285
(Iteration 401 / 1225) loss: 0.840816
(Epoch 2 / 5) train acc: 0.743000; val_acc: 0.578000
(Iteration 501 / 1225) loss: 0.738168
(Iteration 601 / 1225) loss: 0.661739
(Iteration 701 / 1225) loss: 0.808383
(Epoch 3 / 5) train acc: 0.741000; val_acc: 0.539000
(Iteration 801 / 1225) loss: 0.659975
(Iteration 901 / 1225) loss: 0.736018
(Epoch 4 / 5) train acc: 0.751000; val_acc: 0.554000
(Iteration 1001 / 1225) loss: 0.867286
(Iteration 1101 / 1225) loss: 0.786914
(Iteration 1201 / 1225) loss: 0.695123
(Epoch 5 / 5) train acc: 0.784000; val_acc: 0.545000
(Iteration 1 / 1225) loss: 0.861129
(Epoch 0 / 5) train acc: 0.717000; val_acc: 0.568000
(Iteration 101 / 1225) loss: 0.923605
```

```
(Iteration 201 / 1225) loss: 0.934271
(Epoch 1 / 5) train acc: 0.742000; val_acc: 0.553000
(Iteration 301 / 1225) loss: 0.555075
(Iteration 401 / 1225) loss: 0.619434
(Epoch 2 / 5) train acc: 0.788000; val_acc: 0.569000
(Iteration 501 / 1225) loss: 0.559722
(Iteration 601 / 1225) loss: 0.547570
(Iteration 701 / 1225) loss: 0.498987
(Epoch 3 / 5) train acc: 0.834000; val_acc: 0.574000
(Iteration 801 / 1225) loss: 0.496560
(Iteration 901 / 1225) loss: 0.377250
(Epoch 4 / 5) train acc: 0.875000; val_acc: 0.581000
(Iteration 1001 / 1225) loss: 0.415383
(Iteration 1101 / 1225) loss: 0.348312
(Iteration 1201 / 1225) loss: 0.343898
(Epoch 5 / 5) train acc: 0.906000; val_acc: 0.579000
(Iteration 1 / 1225) loss: 0.427988
(Epoch 0 / 5) train acc: 0.831000; val_acc: 0.562000
(Iteration 101 / 1225) loss: 0.881042
(Iteration 201 / 1225) loss: 0.568505
(Epoch 1 / 5) train acc: 0.722000; val_acc: 0.518000
(Iteration 301 / 1225) loss: 0.599686
(Iteration 401 / 1225) loss: 0.407789
(Epoch 2 / 5) train acc: 0.865000; val_acc: 0.565000
(Iteration 501 / 1225) loss: 0.417012
(Iteration 601 / 1225) loss: 0.426047
(Iteration 701 / 1225) loss: 0.409940
(Epoch 3 / 5) train acc: 0.911000; val_acc: 0.569000
(Iteration 801 / 1225) loss: 0.293773
(Iteration 901 / 1225) loss: 0.318903
(Epoch 4 / 5) train acc: 0.912000; val_acc: 0.578000
(Iteration 1001 / 1225) loss: 0.275887
(Iteration 1101 / 1225) loss: 0.280687
(Iteration 1201 / 1225) loss: 0.282292
(Epoch 5 / 5) train acc: 0.924000; val_acc: 0.575000
```

In [32]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

y_test_pred = np.argmax(best_net.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.573
```