knn

February 16, 2021

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 [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

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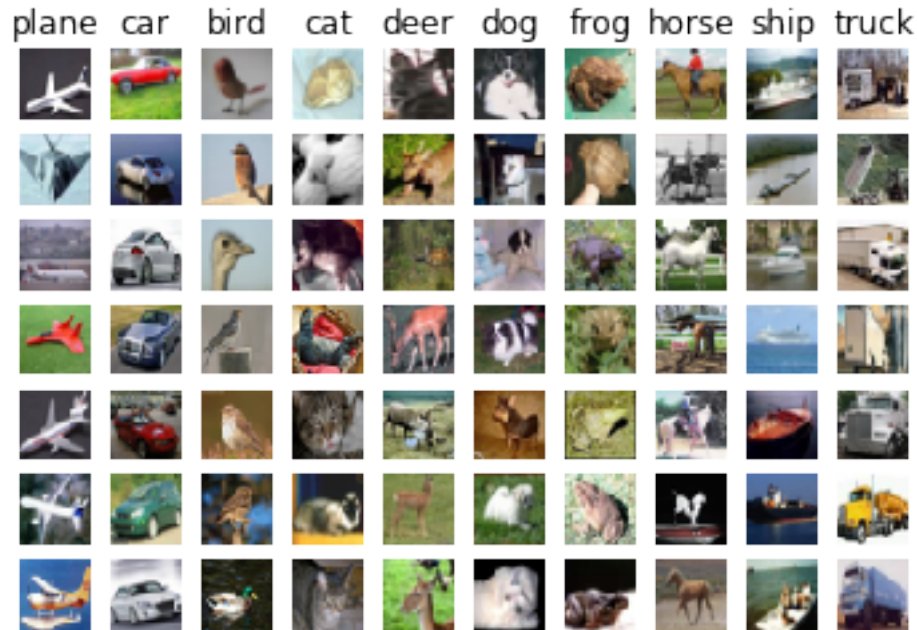
```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

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

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

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

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

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

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

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

```
In [6]: # 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 [7]: # We can visualize the distance matrix: each row is a single test example and
        # its distances to training examples
        plt.imshow(dists, interpolation='none')
        plt.show()
```
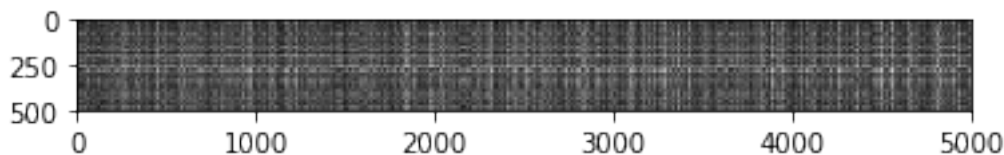

Inline Question 1

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

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

4

Your Answer : * if a test data is very different from most training data, it would look like a bright row. * if a training data is very different from most testing data, it would look like a bright column.

```
In [8]: # 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 [9]: y_test_pred = classifier.predict_labels(dists, k=5)
        num_correct = np.sum(y_test_pred == y_test)
        accuracy = float(num_correct) / num_test
        print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.) 2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean μ and dividing by the standard deviation σ. 4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}. 5. Rotating the coordinate axes of the data.

Your Answer : 1, 2 ,3

Your Explanation : 1. Each data point is subtracted by the meanμ, which is equivalent to a translation of the whole data set in vector space, with no effect on the proximity of the data points.

2. Each data point is subtracted by the pixel mean μ_{ij}, which is equivalent to a spatial transformation of the whole data set, with no effect on the proximity of the data points. 3. Because we are dealing with images, all pixels lie within $[0 - 255]$, the feature ranges are mostly similar, dividing by the standard deviation is equivalent to scaling the data points after translation, all picture do the same scaling, so it would not affect the proximity of the data points. 4. Since we are dividing by the pixel-wise standard deviation σ_{ij}, we are scaling different dimensions differently. The nearest neighbors could thus change and this would in turn affect the performance of the classifier. 5. The L1 distance is more coordinate axes dependent, and rotating the axes will have an effect on the position of the data points, and thus may change the nearest neighbors.

```
In [10]: # 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 [11]: # 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 [12]: # 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 44.786903 seconds
One loop version took 57.939214 seconds
No loop version took 0.171380 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 [13]: 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)*****
```

7

```python
        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:
            classifier = KNearestNeighbor()
            k_to_accuracies[k] = []
            for itr in range(num_folds):
                classifier.train(np.concatenate(np.delete(X_train_folds, itr, 0)),
                            np.concatenate(np.delete(y_train_folds, itr, 0)))
                dists = classifier.compute_distances_no_loops(X_train_folds[itr])
                k_to_accuracies[k].append(np.sum(classifier.predict_labels(dists, k) == y_tra
        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 [14]: # plot the raw observations
         for k in k_choices:
             accuracies = k_to_accuracies[k]
             plt.scatter([k] * len(accuracies), accuracies)
```

```python
# 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()
```
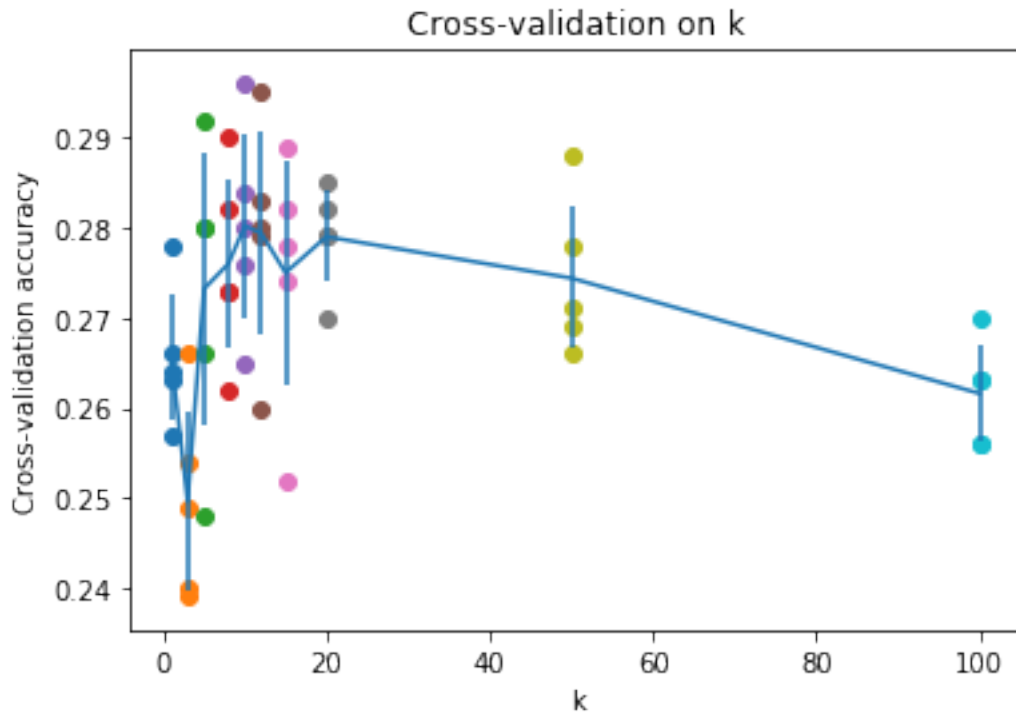

In [15]:
```python
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 1

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))
```

10

```
Got 137 / 500 correct => accuracy: 0.274000
```

Inline Question 3

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

Your Answer : 2, 4

Your Explanation : 1. False. It depends on the distribution of data points, maybe have a circle decision boundary. 2. True. 1-NN will always has correct label which is itself. 3. Flase. It was proved that larger k would be better. 4. True. It is obviously because we need to traverse the entire training set. 5. False.

```
In [ ]:
```

svm

February 16, 2021

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', 'truc
        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)
```
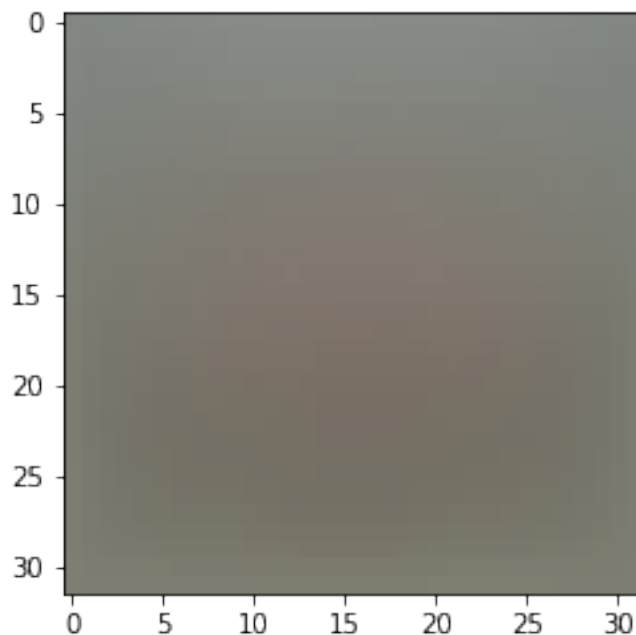
```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



```
(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.

5

```
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: 9.481759
```

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

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

```
In [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: 15.376252 analytic: 15.376252, relative error: 2.006710e-11
numerical: 20.173064 analytic: 20.173064, relative error: 1.715610e-11
numerical: -18.108331 analytic: -18.108331, relative error: 2.933240e-11
numerical: -14.690386 analytic: -14.690386, relative error: 3.530486e-11
numerical: 18.009985 analytic: 18.009985, relative error: 4.923402e-12
numerical: 15.242144 analytic: 15.242144, relative error: 1.277608e-11
numerical: -30.992931 analytic: -30.992931, relative error: 9.336598e-14
numerical: 5.834712 analytic: 5.834712, relative error: 4.345743e-11
numerical: 40.410347 analytic: 40.410347, relative error: 6.520110e-12
numerical: -21.064283 analytic: -21.064283, relative error: 1.990448e-12
```

6

```
numerical: -40.703428 analytic: -40.703428, relative error: 4.940125e-12
numerical: 18.048278 analytic: 18.048278, relative error: 1.254031e-11
numerical: 9.303398 analytic: 9.303398, relative error: 1.529515e-11
numerical: 9.452961 analytic: 9.452961, relative error: 8.587921e-12
numerical: 24.673613 analytic: 24.673613, relative error: 2.456550e-11
numerical: 7.454668 analytic: 7.454668, relative error: 6.709471e-12
numerical: 13.688155 analytic: 13.688155, relative error: 3.061125e-11
numerical: -14.412183 analytic: -14.412183, relative error: 1.996527e-12
numerical: -10.785963 analytic: -10.771950, relative error: 6.499879e-04
numerical: -4.331678 analytic: -4.331678, relative error: 3.939883e-11
```

Inline Question 1

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

Your Answer : A discrepancy may be caused when the loss function is not differentiable at some points. For example, the function $ReLU(x) = max(0, x)$ is not differetiable at x=0. We use the formula $\frac{df(x)}{dx} = \frac{f(x+h)-f(x-h)}{2h}$ to calculate the numerical gradient. Using this formula, the numerical gradients is $f'(\frac{h}{2}) = \frac{3}{4}$, while the analytic gradient is $f'(\frac{h}{2}) = 1$. Thus, a discrtpancy is caused at the point near $x = 0$. Decreasing the step size h can mitigate the margin effect to some extent, or we can consider some special cases, for wxample when $f(x + h) > 0 \quad and \quad f(x - h)$, we just set $f'(x) = 1$ when $x > 0$ and $f'(x) = 0$ otherwise.

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

         from cs231n.classifiers.linear_svm import svm_loss_vectorized
         tic = time.time()
         loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('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: 9.481759e+00 computed in 0.121874s
Vectorized loss: 9.481759e+00 computed in 0.002630s
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 0.128448s
Vectorized loss and gradient: computed in 0.002032s
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 782.578832
iteration 100 / 1500: loss 284.482432
iteration 200 / 1500: loss 106.943416
iteration 300 / 1500: loss 41.991659
iteration 400 / 1500: loss 19.040192
iteration 500 / 1500: loss 10.333565
iteration 600 / 1500: loss 6.850206
iteration 700 / 1500: loss 6.146433
iteration 800 / 1500: loss 5.372751
```

```
iteration 900 / 1500: loss 5.404740
iteration 1000 / 1500: loss 5.519727
iteration 1100 / 1500: loss 5.269093
iteration 1200 / 1500: loss 5.505828
iteration 1300 / 1500: loss 4.919089
iteration 1400 / 1500: loss 5.436292
That took 4.222337s
```

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()
```
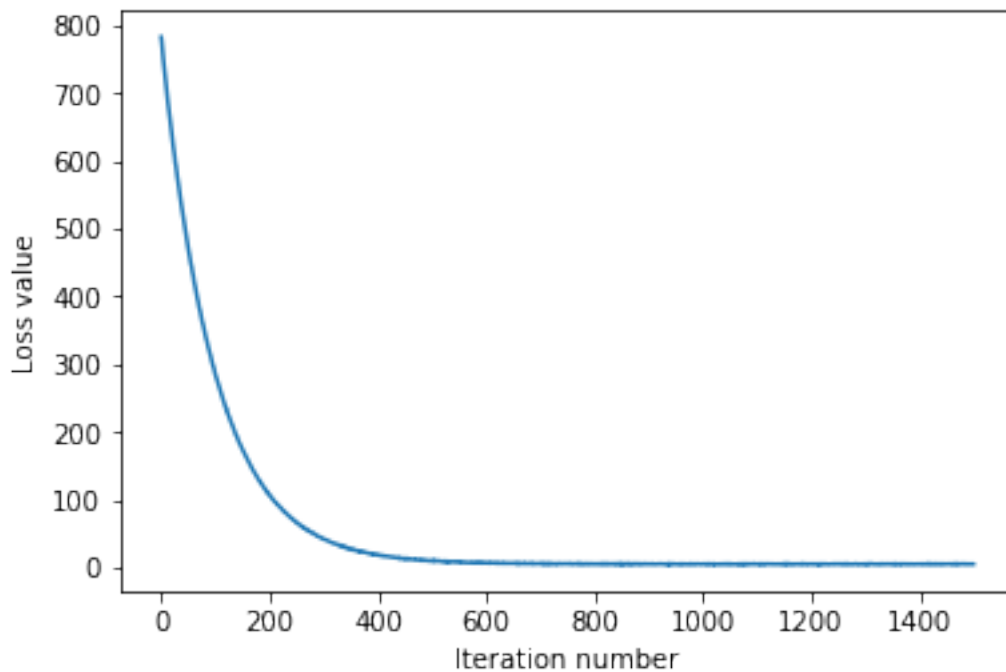

In [14]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.366510
validation accuracy: 0.381000
```

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

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

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

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

          # Provided as a reference. You may or may not want to change these hyperparameters
          learning_rates = np.linspace(3e-7, 5e-7, 5)
          regularization_strengths = np.linspace(5e3, 6e3, 5)

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

          for rs in regularization_strengths:
              for lr in learning_rates:
                  svm = LinearSVM()
                  svm.train(X_train, y_train, learning_rate=lr, reg=rs, num_iters=2000)
                  y_train_pred = svm.predict(X_train)
                  train_accuracy = np.mean(y_train == y_train_pred)
                  y_val_pred = svm.predict(X_val)
                  val_accuracy = np.mean(y_val == y_val_pred)
                  results[(lr, rs)] = train_accuracy, val_accuracy
                  if val_accuracy > best_val:
                      best_val = val_accuracy
```

10

```
                    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 during cross-validation: %f' % best_val)

lr 3.000000e-07 reg 5.000000e+03 train accuracy: 0.376612 val accuracy: 0.372000
lr 3.000000e-07 reg 5.250000e+03 train accuracy: 0.381612 val accuracy: 0.376000
lr 3.000000e-07 reg 5.500000e+03 train accuracy: 0.378592 val accuracy: 0.381000
lr 3.000000e-07 reg 5.750000e+03 train accuracy: 0.385408 val accuracy: 0.379000
lr 3.000000e-07 reg 6.000000e+03 train accuracy: 0.375041 val accuracy: 0.388000
lr 3.500000e-07 reg 5.000000e+03 train accuracy: 0.374327 val accuracy: 0.382000
lr 3.500000e-07 reg 5.250000e+03 train accuracy: 0.384429 val accuracy: 0.409000
lr 3.500000e-07 reg 5.500000e+03 train accuracy: 0.374102 val accuracy: 0.401000
lr 3.500000e-07 reg 5.750000e+03 train accuracy: 0.366327 val accuracy: 0.380000
lr 3.500000e-07 reg 6.000000e+03 train accuracy: 0.368000 val accuracy: 0.388000
lr 4.000000e-07 reg 5.000000e+03 train accuracy: 0.352306 val accuracy: 0.369000
lr 4.000000e-07 reg 5.250000e+03 train accuracy: 0.376735 val accuracy: 0.399000
lr 4.000000e-07 reg 5.500000e+03 train accuracy: 0.380082 val accuracy: 0.382000
lr 4.000000e-07 reg 5.750000e+03 train accuracy: 0.371592 val accuracy: 0.379000
lr 4.000000e-07 reg 6.000000e+03 train accuracy: 0.373653 val accuracy: 0.374000
lr 4.500000e-07 reg 5.000000e+03 train accuracy: 0.368531 val accuracy: 0.395000
lr 4.500000e-07 reg 5.250000e+03 train accuracy: 0.370224 val accuracy: 0.379000
lr 4.500000e-07 reg 5.500000e+03 train accuracy: 0.362776 val accuracy: 0.361000
lr 4.500000e-07 reg 5.750000e+03 train accuracy: 0.359184 val accuracy: 0.366000
lr 4.500000e-07 reg 6.000000e+03 train accuracy: 0.372041 val accuracy: 0.376000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.361245 val accuracy: 0.379000
lr 5.000000e-07 reg 5.250000e+03 train accuracy: 0.345878 val accuracy: 0.350000
lr 5.000000e-07 reg 5.500000e+03 train accuracy: 0.368939 val accuracy: 0.373000
lr 5.000000e-07 reg 5.750000e+03 train accuracy: 0.369286 val accuracy: 0.375000
lr 5.000000e-07 reg 6.000000e+03 train accuracy: 0.350714 val accuracy: 0.363000
best validation accuracy achieved during cross-validation: 0.409000


In [24]: # Visualize the cross-validation results
         import math
         import pdb

         # pdb.set_trace()
```

```python
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()
```
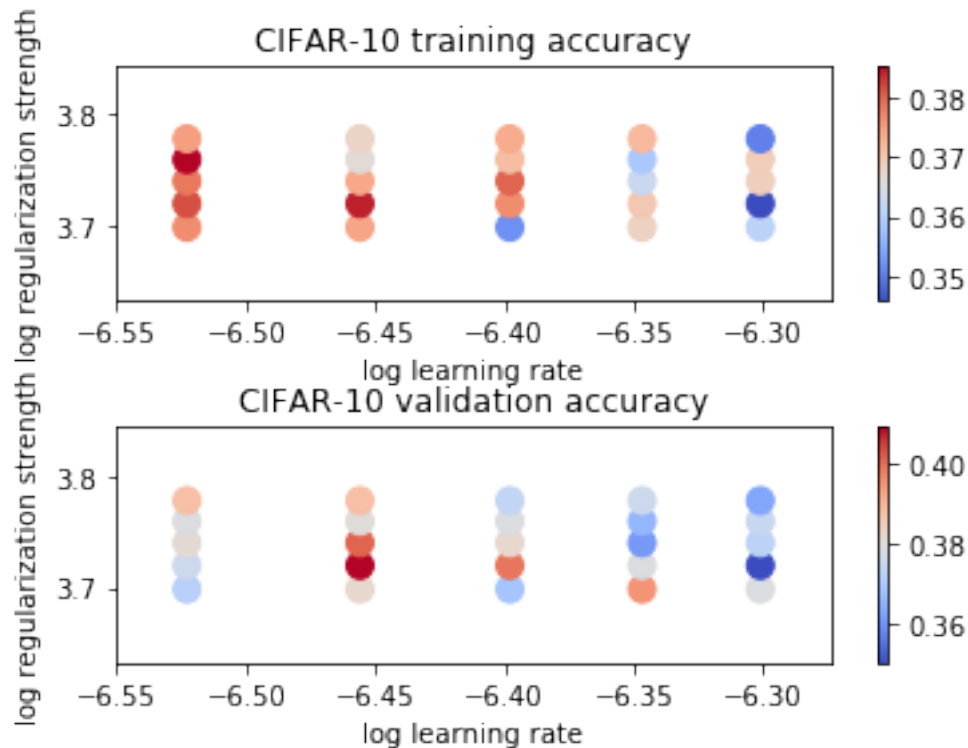
```
In [25]: # Evaluate the best svm on test set
         y_test_pred = best_svm.predict(X_test)
         test_accuracy = np.mean(y_test == y_test_pred)
         print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)

linear SVM on raw pixels final test set accuracy: 0.377000


In [26]: # 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 they way that they do.

Your Answer : The SVM weights look like the averaged result of all the images belonging to this class, for example, the weight of a horse can roughly see the body and legs of the horse, but

with two heads. Since we get the score by computing the dot product, the learning direction of the weights is to compute a relatively high score for each picture of this class, and the result of this learning is such an averaging-like picture.

In [ ]:

softmax

February 16, 2021

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 [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
```

```
In [2]: 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'
```

```python
    # Cleaning up variables to prevent loading data multiple times (which may cause mer
    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 [6]: # 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: 2.333019
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : Because our weight matrix W is randomly initialized, the scores of each class should be uniformly distributed after the first epoch, so the probability of each class is around 1/10, where 10 is the number of classes, so the loss is close to -log(0.1).

```
In [7]:  # 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: -2.066922 analytic: -2.066922, relative error: 1.461571e-08
numerical: -1.867313 analytic: -1.867314, relative error: 4.974683e-08
numerical: 1.551848 analytic: 1.551848, relative error: 8.132535e-09
numerical: 0.495528 analytic: 0.495527, relative error: 1.753607e-07
numerical: 1.037626 analytic: 1.037626, relative error: 1.450010e-08
numerical: -1.686654 analytic: -1.686654, relative error: 1.906742e-08
numerical: -0.248827 analytic: -0.248827, relative error: 1.810776e-07
numerical: 0.068697 analytic: 0.068697, relative error: 3.029219e-07
numerical: 0.402182 analytic: 0.402182, relative error: 1.625887e-07
numerical: -2.225517 analytic: -2.225517, relative error: 2.865967e-09
numerical: 0.592345 analytic: 0.592345, relative error: 1.532042e-07
numerical: -0.760181 analytic: -0.760181, relative error: 2.895375e-08
numerical: 0.521520 analytic: 0.521520, relative error: 4.009850e-12
numerical: -0.887937 analytic: -0.887937, relative error: 3.858837e-08
numerical: 1.585639 analytic: 1.585639, relative error: 1.527680e-08
numerical: 2.962142 analytic: 2.962142, relative error: 1.807236e-08
numerical: -0.345628 analytic: -0.345628, relative error: 3.150321e-07
numerical: -3.549996 analytic: -3.549996, relative error: 1.152965e-08
numerical: 0.586687 analytic: 0.586687, relative error: 1.132797e-07
numerical: -1.752683 analytic: -1.752683, relative error: 1.205578e-08

```
In [10]:  # Now that we have a naive implementation of the softmax loss function and its gradie
          # 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()
```

4

```
        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: 2.333019e+00 computed in 0.165586s
vectorized loss: 2.333019e+00 computed in 0.003064s
Loss difference: 0.000000
Gradient difference: 0.000000
```

```
In [19]: # 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.linspace(6e-7, 9e-7, 5)
         regularization_strengths = np.linspace(4e3, 6e3, 5)

         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
         for rs in regularization_strengths:
             for lr in learning_rates:
                 softmax = Softmax()
                 softmax.train(X_train, y_train, learning_rate=lr, reg=rs, num_iters=2000)
                 y_train_pred = softmax.predict(X_train)
                 train_accuracy = np.mean(y_train == y_train_pred)
                 y_val_pred = softmax.predict(X_val)
                 val_accuracy = np.mean(y_val == y_val_pred)
                 results[(lr, rs)] = train_accuracy, val_accuracy
                 if val_accuracy > best_val:
```

```python
                best_val = val_accuracy
                best_softmax = softmax
        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)
```

```
lr 6.000000e-07 reg 4.000000e+03 train accuracy: 0.371102 val accuracy: 0.367000
lr 6.000000e-07 reg 4.500000e+03 train accuracy: 0.373857 val accuracy: 0.377000
lr 6.000000e-07 reg 5.000000e+03 train accuracy: 0.367224 val accuracy: 0.382000
lr 6.000000e-07 reg 5.500000e+03 train accuracy: 0.362980 val accuracy: 0.379000
lr 6.000000e-07 reg 6.000000e+03 train accuracy: 0.362347 val accuracy: 0.368000
lr 6.750000e-07 reg 4.000000e+03 train accuracy: 0.368735 val accuracy: 0.383000
lr 6.750000e-07 reg 4.500000e+03 train accuracy: 0.371327 val accuracy: 0.384000
lr 6.750000e-07 reg 5.000000e+03 train accuracy: 0.367061 val accuracy: 0.377000
lr 6.750000e-07 reg 5.500000e+03 train accuracy: 0.365388 val accuracy: 0.380000
lr 6.750000e-07 reg 6.000000e+03 train accuracy: 0.370163 val accuracy: 0.378000
lr 7.500000e-07 reg 4.000000e+03 train accuracy: 0.379102 val accuracy: 0.387000
lr 7.500000e-07 reg 4.500000e+03 train accuracy: 0.374776 val accuracy: 0.386000
lr 7.500000e-07 reg 5.000000e+03 train accuracy: 0.363408 val accuracy: 0.376000
lr 7.500000e-07 reg 5.500000e+03 train accuracy: 0.366918 val accuracy: 0.370000
lr 7.500000e-07 reg 6.000000e+03 train accuracy: 0.359939 val accuracy: 0.366000
lr 8.250000e-07 reg 4.000000e+03 train accuracy: 0.371531 val accuracy: 0.391000
lr 8.250000e-07 reg 4.500000e+03 train accuracy: 0.368429 val accuracy: 0.367000
lr 8.250000e-07 reg 5.000000e+03 train accuracy: 0.372776 val accuracy: 0.392000
lr 8.250000e-07 reg 5.500000e+03 train accuracy: 0.369306 val accuracy: 0.378000
lr 8.250000e-07 reg 6.000000e+03 train accuracy: 0.364939 val accuracy: 0.372000
lr 9.000000e-07 reg 4.000000e+03 train accuracy: 0.371041 val accuracy: 0.387000
lr 9.000000e-07 reg 4.500000e+03 train accuracy: 0.374796 val accuracy: 0.387000
lr 9.000000e-07 reg 5.000000e+03 train accuracy: 0.369735 val accuracy: 0.391000
lr 9.000000e-07 reg 5.500000e+03 train accuracy: 0.367224 val accuracy: 0.379000
lr 9.000000e-07 reg 6.000000e+03 train accuracy: 0.355510 val accuracy: 0.358000
best validation accuracy achieved during cross-validation: 0.392000
```

```python
In [21]: # 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: 0.382000
```

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 : It is possible.

YourExplanation : Because in SVM we have $L_i = \max(0, s_j - s_{y_i} + \Delta)$, so L_i is possivle to be zero. But in Softmax, we have $L_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$, the denominator must not be 0, and generally we have more than one class, the numerators and denominators are not equal, so L_i will not be equal to 0.

```python
In [22]: # 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', 'tr
         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])
```



```python
In [ ]:
```

two_layer_net

February 16, 2021

1 Implementing a Neural Network

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

```
In [1]: # A bit of setup

        import numpy as np
        import matplotlib.pyplot as plt

        from cs231n.classifiers.neural_net import TwoLayerNet

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

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

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

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

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

        input_size = 4
        hidden_size = 10
        num_classes = 3
```

```
num_inputs = 5

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

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

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

2   Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

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

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

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

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

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

Difference between your scores and correct scores:
3.6802720745909845e-08
```

3 Forward pass: compute loss

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

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

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

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

4 Backward pass

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

```
In [9]: from cs231n.gradient_check import import eval_numerical_gradient

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

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

        # these should all be less than 1e-8 or so
        for param_name in grads:
            f = lambda W: net.loss(X, y, reg=0.05)[0]
            param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[pa
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447625e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

5 Train the network

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

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

```python
In [10]: net = init_toy_model()
         stats = net.train(X, y, X, y,
                     learning_rate=1e-1, reg=5e-6,
                     num_iters=100, verbose=False)

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

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

Final training loss:  0.017149607938732093
```

Training Loss history
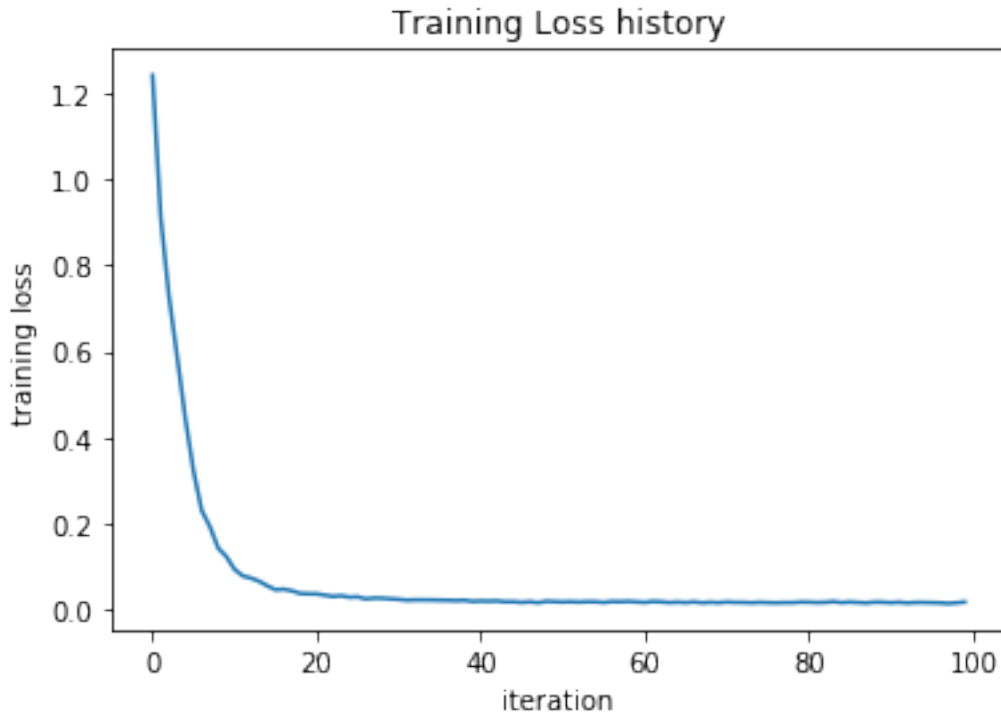
6 Load the data

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

```
In [11]: from cs231n.data_utils import load_CIFAR10

         def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
             """
             Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
             it for the two-layer neural net classifier. These are the same steps as
             we used for the SVM, but condensed to a single function.
             """
             # Load the raw CIFAR-10 data
             cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

```python
            pass

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

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

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

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

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


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

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

7 Train a network

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

```
In [12]: input_size = 32 * 32 * 3
         hidden_size = 50
         num_classes = 10
         net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

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

8 Debug the training

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

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

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

```
In [13]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(stats['loss_history'])
         plt.title('Loss history')
```

```
plt.xlabel('Iteration')
plt.ylabel('Loss')

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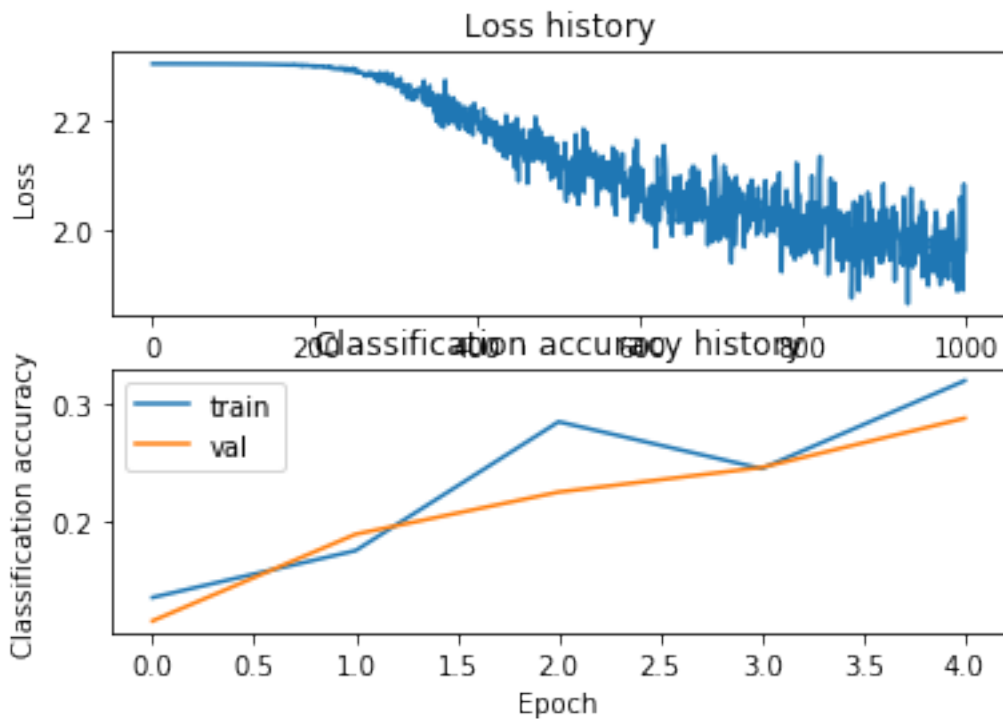

In [14]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

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

show_net_weights(net)
```
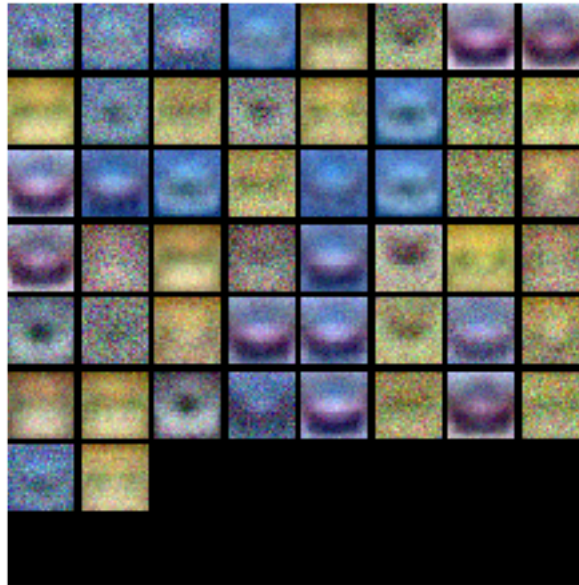
9 Tune your hyperparameters

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

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

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

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

    Explain your hyperparameter tuning process below.

    Your Answer : First, we set num_iter=500, learning_rate_decay=0.95, and tried with the following parameters * hidden_size = [60, 96, 132] * learning_rates = np.linspace(1e-3, 9e-3, 5) * regularization_strengths = [0.7, 0.9, 1, 1.2]

I found that we got better results when the hidden_size was larger, and we got better results when the learning rate and reg were smaller, so we narrowed the range and tried a second time with the following parameters * hidden_size = [100, 110, 120, 130, 140, 150] * learning_rates = np.linspace(7e-4, 1.2e-3, 6) * regularization_strengths = np.linspace(0.1, 0.7, 7)

Then found that reg in 0.1,0.5 will have good performance, 0.3 occasionally also has good performance, learning rate in the vicinity of 1.1 will have good performance, hidden_size in about 130 better performance, continue to narrow the range, with the following parameters to try * hidden_size = [100,110,120, 130, 140] * learning_rates = np.linspace(1e-3, 1.2e-3, 3) * regularization_strengths = np.linspace(0.1, 0.7, 4)

Then I found no big improvement, so I raised num_iter and ran it again with the same parameters, then get the final result

```python
In [25]: best_net = None # store the best model into this

        #############################################################################
        # TODO: Tune hyperparameters using the validation set. Store your best trained  #
        # model in best_net.                                                           #
        #                                                                              #
        # To help debug your network, it may help to use visualizations similar to the #
        # ones we used above; these visualizations will have significant qualitative   #
        # differences from the ones we saw above for the poorly tuned network.         #
        #                                                                              #
        # Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
        # write code to sweep through possible combinations of hyperparameters         #
        # automatically like we did on the previous exercises.                         #
        #############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        hidden_size = [100,110,120, 130, 140]
        learning_rates = np.linspace(1e-3, 1.2e-3, 3)
        regularization_strengths = np.linspace(0.1, 0.7, 4)
        results = {}
        best_val = 0

        for hs in hidden_size:
            for reg in regularization_strengths:
                for lr in learning_rates:
                    net = TwoLayerNet(input_size, hs, num_classes)
                    net.train(X_train, y_train, X_val, y_val,
                    num_iters=2000, batch_size=200,
                    learning_rate=lr, learning_rate_decay=0.95,
                    reg= reg, verbose=False)

        #            y_train_pred = svm.predict(X_train)
        #            train_accuracy = np.mean(y_train == y_train_pred)
                    y_val_pred = net.predict(X_val)
                    val_accuracy = np.mean(y_val == y_val_pred)
                    results[(hs,lr, reg)] = val_accuracy
```

10

```
                if val_accuracy > best_val:
                    best_val = val_accuracy
                    best_net = net
        pass
        # Print out results.
        for hs, lr, reg in sorted(results):
            val_accuracy = results[(hs, lr, reg)]
            print('hs %e lr %e reg %e  val accuracy: %f' % (
                    hs, lr, reg, val_accuracy))

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

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

```
hs 1.000000e+02 lr 1.000000e-03 reg 1.000000e-01  val accuracy: 0.514000
hs 1.000000e+02 lr 1.000000e-03 reg 3.000000e-01  val accuracy: 0.517000
hs 1.000000e+02 lr 1.000000e-03 reg 5.000000e-01  val accuracy: 0.484000
hs 1.000000e+02 lr 1.000000e-03 reg 7.000000e-01  val accuracy: 0.487000
hs 1.000000e+02 lr 1.100000e-03 reg 1.000000e-01  val accuracy: 0.516000
hs 1.000000e+02 lr 1.100000e-03 reg 3.000000e-01  val accuracy: 0.499000
hs 1.000000e+02 lr 1.100000e-03 reg 5.000000e-01  val accuracy: 0.498000
hs 1.000000e+02 lr 1.100000e-03 reg 7.000000e-01  val accuracy: 0.501000
hs 1.000000e+02 lr 1.200000e-03 reg 1.000000e-01  val accuracy: 0.516000
hs 1.000000e+02 lr 1.200000e-03 reg 3.000000e-01  val accuracy: 0.493000
hs 1.000000e+02 lr 1.200000e-03 reg 5.000000e-01  val accuracy: 0.503000
hs 1.000000e+02 lr 1.200000e-03 reg 7.000000e-01  val accuracy: 0.507000
hs 1.100000e+02 lr 1.000000e-03 reg 1.000000e-01  val accuracy: 0.494000
hs 1.100000e+02 lr 1.000000e-03 reg 3.000000e-01  val accuracy: 0.506000
hs 1.100000e+02 lr 1.000000e-03 reg 5.000000e-01  val accuracy: 0.496000
hs 1.100000e+02 lr 1.000000e-03 reg 7.000000e-01  val accuracy: 0.495000
hs 1.100000e+02 lr 1.100000e-03 reg 1.000000e-01  val accuracy: 0.506000
hs 1.100000e+02 lr 1.100000e-03 reg 3.000000e-01  val accuracy: 0.502000
hs 1.100000e+02 lr 1.100000e-03 reg 5.000000e-01  val accuracy: 0.508000
hs 1.100000e+02 lr 1.100000e-03 reg 7.000000e-01  val accuracy: 0.493000
hs 1.100000e+02 lr 1.200000e-03 reg 1.000000e-01  val accuracy: 0.517000
hs 1.100000e+02 lr 1.200000e-03 reg 3.000000e-01  val accuracy: 0.506000
hs 1.100000e+02 lr 1.200000e-03 reg 5.000000e-01  val accuracy: 0.508000
hs 1.100000e+02 lr 1.200000e-03 reg 7.000000e-01  val accuracy: 0.487000
hs 1.200000e+02 lr 1.000000e-03 reg 1.000000e-01  val accuracy: 0.504000
hs 1.200000e+02 lr 1.000000e-03 reg 3.000000e-01  val accuracy: 0.508000
hs 1.200000e+02 lr 1.000000e-03 reg 5.000000e-01  val accuracy: 0.518000
hs 1.200000e+02 lr 1.000000e-03 reg 7.000000e-01  val accuracy: 0.482000
hs 1.200000e+02 lr 1.100000e-03 reg 1.000000e-01  val accuracy: 0.501000
hs 1.200000e+02 lr 1.100000e-03 reg 3.000000e-01  val accuracy: 0.495000
hs 1.200000e+02 lr 1.100000e-03 reg 5.000000e-01  val accuracy: 0.510000
hs 1.200000e+02 lr 1.100000e-03 reg 7.000000e-01  val accuracy: 0.483000
hs 1.200000e+02 lr 1.200000e-03 reg 1.000000e-01  val accuracy: 0.513000
hs 1.200000e+02 lr 1.200000e-03 reg 3.000000e-01  val accuracy: 0.507000
```

```
hs 1.200000e+02 lr 1.200000e-03 reg 5.000000e-01  val accuracy: 0.487000
hs 1.200000e+02 lr 1.200000e-03 reg 7.000000e-01  val accuracy: 0.481000
hs 1.300000e+02 lr 1.000000e-03 reg 1.000000e-01  val accuracy: 0.516000
hs 1.300000e+02 lr 1.000000e-03 reg 3.000000e-01  val accuracy: 0.503000
hs 1.300000e+02 lr 1.000000e-03 reg 5.000000e-01  val accuracy: 0.496000
hs 1.300000e+02 lr 1.000000e-03 reg 7.000000e-01  val accuracy: 0.483000
hs 1.300000e+02 lr 1.100000e-03 reg 1.000000e-01  val accuracy: 0.507000
hs 1.300000e+02 lr 1.100000e-03 reg 3.000000e-01  val accuracy: 0.512000
hs 1.300000e+02 lr 1.100000e-03 reg 5.000000e-01  val accuracy: 0.496000
hs 1.300000e+02 lr 1.100000e-03 reg 7.000000e-01  val accuracy: 0.507000
hs 1.300000e+02 lr 1.200000e-03 reg 1.000000e-01  val accuracy: 0.500000
hs 1.300000e+02 lr 1.200000e-03 reg 3.000000e-01  val accuracy: 0.508000
hs 1.300000e+02 lr 1.200000e-03 reg 5.000000e-01  val accuracy: 0.520000
hs 1.300000e+02 lr 1.200000e-03 reg 7.000000e-01  val accuracy: 0.495000
hs 1.400000e+02 lr 1.000000e-03 reg 1.000000e-01  val accuracy: 0.504000
hs 1.400000e+02 lr 1.000000e-03 reg 3.000000e-01  val accuracy: 0.503000
hs 1.400000e+02 lr 1.000000e-03 reg 5.000000e-01  val accuracy: 0.509000
hs 1.400000e+02 lr 1.000000e-03 reg 7.000000e-01  val accuracy: 0.498000
hs 1.400000e+02 lr 1.100000e-03 reg 1.000000e-01  val accuracy: 0.496000
hs 1.400000e+02 lr 1.100000e-03 reg 3.000000e-01  val accuracy: 0.506000
hs 1.400000e+02 lr 1.100000e-03 reg 5.000000e-01  val accuracy: 0.488000
hs 1.400000e+02 lr 1.100000e-03 reg 7.000000e-01  val accuracy: 0.501000
hs 1.400000e+02 lr 1.200000e-03 reg 1.000000e-01  val accuracy: 0.495000
hs 1.400000e+02 lr 1.200000e-03 reg 3.000000e-01  val accuracy: 0.519000
hs 1.400000e+02 lr 1.200000e-03 reg 5.000000e-01  val accuracy: 0.485000
hs 1.400000e+02 lr 1.200000e-03 reg 7.000000e-01  val accuracy: 0.483000
best validation accuracy achieved during cross-validation: 0.520000
```

In [26]: # Print your validation accuracy: this should be above 48%
         val_acc = (best_net.predict(X_val) == y_val).mean()
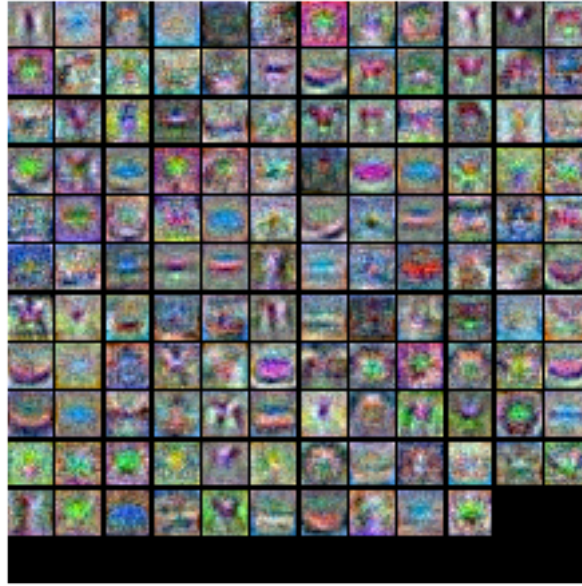   print('Validation accuracy: ', val_acc)

Validation accuracy: 0.52

In [27]: # Visualize the weights of the best network
         show_net_weights(best_net)

10   Run on the test set

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

```
In [28]: # Print your test accuracy: this should be above 48%
         test_acc = (best_net.predict(X_test) == y_test).mean()
         print('Test accuracy: ', test_acc)

Test accuracy:  0.509
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

$Your Answer:$ 1, 3

$Your Explanation:$ 1. True.training on a larger dataset help you extract more features from dataset, it can avoid overfitting. 2. False. More hidden unit means more complexity of the model,

13

it may exacerbate overfitting. 3. True. Regularization reduces the complexity of the model by reducing the dependency of the model on some useless features, which can help to avoid overfitting.

In [ ]:

features

February 16, 2021

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


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

            # Cleaning up variables to prevent loading data multiple times (which may cause mer
```

1

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

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

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

```python
        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.

```python
In [5]: # Use the validation set to tune the learning rate and regularization strength

        from cs231n.classifiers.linear_classifier import LinearSVM

        # learning_rates = [1e-9, 1e-8, 1e-7]
        # regularization_strengths = [5e4, 5e5, 5e6]
        learning_rates = np.linspace(3e-7, 5e-7, 5)
        regularization_strengths = np.linspace(5e3, 6e3, 5)

        results = {}
        best_val = -1
        best_svm = None

        ################################################################################
        # TODO:                                                                        #
        # Use the validation set to set the learning rate and regularization strength. #
        # This should be identical to the validation that you did for the SVM; save    #
        # the best trained classifer in best_svm. You might also want to play          #
        # with different numbers of bins in the color histogram. If you are careful    #
        # you should be able to get accuracy of near 0.44 on the validation set.       #
        ################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        for rs in regularization_strengths:
```

```python
        for lr in learning_rates:
            svm = LinearSVM()
            svm.train(X_train_feats, y_train, learning_rate=lr, reg=rs, num_iters=2000)
            y_train_pred = svm.predict(X_train_feats)
            train_accuracy = np.mean(y_train == y_train_pred)
            y_val_pred = svm.predict(X_val_feats)
            val_accuracy = np.mean(y_val == y_val_pred)
            results[(lr, rs)] = train_accuracy, val_accuracy
            if val_accuracy > best_val:
                best_val = val_accuracy
                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 during cross-validation: %f' % best_val)
```

```
lr 3.000000e-07 reg 5.000000e+03 train accuracy: 0.411653 val accuracy: 0.413000
lr 3.000000e-07 reg 5.250000e+03 train accuracy: 0.415714 val accuracy: 0.416000
lr 3.000000e-07 reg 5.500000e+03 train accuracy: 0.415020 val accuracy: 0.414000
lr 3.000000e-07 reg 5.750000e+03 train accuracy: 0.414204 val accuracy: 0.415000
lr 3.000000e-07 reg 6.000000e+03 train accuracy: 0.414408 val accuracy: 0.420000
lr 3.500000e-07 reg 5.000000e+03 train accuracy: 0.416633 val accuracy: 0.421000
lr 3.500000e-07 reg 5.250000e+03 train accuracy: 0.415000 val accuracy: 0.421000
lr 3.500000e-07 reg 5.500000e+03 train accuracy: 0.415633 val accuracy: 0.416000
lr 3.500000e-07 reg 5.750000e+03 train accuracy: 0.417286 val accuracy: 0.417000
lr 3.500000e-07 reg 6.000000e+03 train accuracy: 0.412898 val accuracy: 0.415000
lr 4.000000e-07 reg 5.000000e+03 train accuracy: 0.417388 val accuracy: 0.431000
lr 4.000000e-07 reg 5.250000e+03 train accuracy: 0.416694 val accuracy: 0.420000
lr 4.000000e-07 reg 5.500000e+03 train accuracy: 0.415714 val accuracy: 0.418000
lr 4.000000e-07 reg 5.750000e+03 train accuracy: 0.416020 val accuracy: 0.418000
lr 4.000000e-07 reg 6.000000e+03 train accuracy: 0.415122 val accuracy: 0.415000
lr 4.500000e-07 reg 5.000000e+03 train accuracy: 0.414673 val accuracy: 0.423000
lr 4.500000e-07 reg 5.250000e+03 train accuracy: 0.417245 val accuracy: 0.422000
lr 4.500000e-07 reg 5.500000e+03 train accuracy: 0.412551 val accuracy: 0.412000
lr 4.500000e-07 reg 5.750000e+03 train accuracy: 0.414857 val accuracy: 0.420000
lr 4.500000e-07 reg 6.000000e+03 train accuracy: 0.415571 val accuracy: 0.424000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.413878 val accuracy: 0.427000
lr 5.000000e-07 reg 5.250000e+03 train accuracy: 0.415122 val accuracy: 0.416000
lr 5.000000e-07 reg 5.500000e+03 train accuracy: 0.413020 val accuracy: 0.423000
lr 5.000000e-07 reg 5.750000e+03 train accuracy: 0.416265 val accuracy: 0.418000
lr 5.000000e-07 reg 6.000000e+03 train accuracy: 0.414816 val accuracy: 0.418000
```

best validation accuracy achieved during cross-validation: 0.431000

In [6]: # 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)

0.424

In [7]: # An important way to gain intuition about how an algorithm works is to
        # visualize the mistakes that it makes. In this visualization, we show examples
        # of images that are misclassified by our current system. The first column
        # shows images that our system labeled as "plane" but whose true label is
        # something other than "plane".

        examples_per_class = 8
        classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', '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 images that were misclassified all had elements similar to the plane, such as a metallic appearance, monotone colors, and images with more sky or a blue color similar to the sky.

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 [19]: # 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 [20]: from cs231n.classifiers.neural_net import TwoLayerNet

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

         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_rates = [1e-1, 5e-1, 0.8]
```

```python
        regularization_strengths = [2.5e-4, 1e-3, 5e-3, 1e-2, 1e-1, 0.5]
        results = {}
        best_val = 0


        for reg in regularization_strengths:
            for lr in learning_rates:
                net = TwoLayerNet(input_dim, hidden_dim, num_classes)
                net.train(X_train_feats, y_train, X_val_feats, y_val,
                num_iters=2000, batch_size=200,
                learning_rate=lr, learning_rate_decay=0.95,
                reg= reg, verbose=False)

        #            y_train_pred = svm.predict(X_train)
        #            train_accuracy = np.mean(y_train == y_train_pred)
                y_val_pred = net.predict(X_val_feats)
                val_accuracy = np.mean(y_val == y_val_pred)
                results[(lr, reg)] = val_accuracy
                if val_accuracy > best_val:
                    best_val = val_accuracy
                    best_net = net
        pass
        # Print out results.
        for lr, reg in sorted(results):
            val_accuracy = results[(lr, reg)]
            print('lr %e reg %e  val accuracy: %f' % (
                        lr, reg, val_accuracy))

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

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

lr 1.000000e-01 reg 2.500000e-04  val accuracy: 0.530000
lr 1.000000e-01 reg 1.000000e-03  val accuracy: 0.530000
lr 1.000000e-01 reg 5.000000e-03  val accuracy: 0.516000
lr 1.000000e-01 reg 1.000000e-02  val accuracy: 0.512000
lr 1.000000e-01 reg 1.000000e-01  val accuracy: 0.295000
lr 1.000000e-01 reg 5.000000e-01  val accuracy: 0.107000
lr 5.000000e-01 reg 2.500000e-04  val accuracy: 0.592000
lr 5.000000e-01 reg 1.000000e-03  val accuracy: 0.588000
lr 5.000000e-01 reg 5.000000e-03  val accuracy: 0.542000
lr 5.000000e-01 reg 1.000000e-02  val accuracy: 0.513000
lr 5.000000e-01 reg 1.000000e-01  val accuracy: 0.282000
lr 5.000000e-01 reg 5.000000e-01  val accuracy: 0.078000
lr 8.000000e-01 reg 2.500000e-04  val accuracy: 0.569000
lr 8.000000e-01 reg 1.000000e-03  val accuracy: 0.585000
lr 8.000000e-01 reg 5.000000e-03  val accuracy: 0.534000
```

```
lr 8.000000e-01 reg 1.000000e-02  val accuracy: 0.484000
lr 8.000000e-01 reg 1.000000e-01  val accuracy: 0.289000
lr 8.000000e-01 reg 5.000000e-01  val accuracy: 0.098000
best validation accuracy achieved during cross-validation: 0.592000
```

In [21]: `# Run your best neural net classifier on the test set. You should be able`
`# to get more than 55% accuracy.`

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

```
0.581
```

In [ ]: