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.

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.

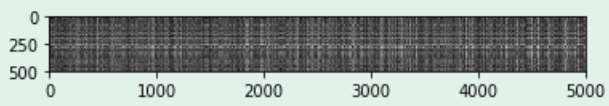
In [6]:

```
# 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?

YourAnswer: *fill this in.*

1. This test point is very different from most samples in the trainning data set. It is probably from another class.
2. This trainning data point is very different from all test points.

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.

YourAnswer:

1,2,3

YourExplanation:

For L1 distance, the difference between two values will not change when both of the values subtract the same number, no matter which kind of mean that number is, so 1, 2 are correct. $|| x^{(i)} - x^{(j)} || = || (x^{(i)} - \mu) - (x^{(j)} - \mu) ||$

3 is just normalizing the data, the ordering of distance will not change after normalizing. So 3 is also correct.

4 is wrong because the pixel-wise standard deviation is different for each pixel and the relative magnitude of the sum of difference will change. The performance of L1 distance will change.

5 is wrong because L1 distance will change when rotating the coordinate axes of the data. For example, there are 3 points x(1,0),

y(0,1), z(3,0). The L1 distances between x, y is 2, the L1 distances between x, z is 2. They are the same.

Now rotate the axes clockwisely by 45 degrees, the L1 distances between x, y becomes $\sqrt{2}$, the L1 distances between x, z is $2\sqrt{2}$. So 5 is wrong.

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

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

```
Two loop version took 39.371335 seconds
One loop version took 93.376822 seconds
No loop version took 0.643046 seconds
```

Cross-validation

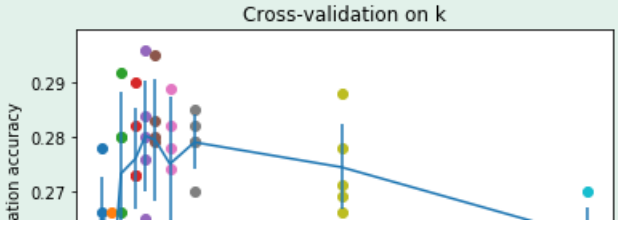
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

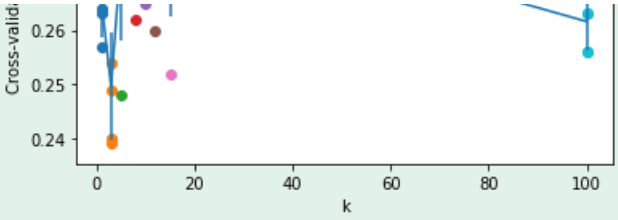
In [14]:

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

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

X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

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

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


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

for k in k_choices:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        # All elemetns except the i-th one should be concatenated to be the train set
        X_train_set = np.concatenate(X_train_folds[0: i] + X_train_folds[i + 1: num_folds])
        y_train_set = np.concatenate(y_train_folds[0: i] + y_train_folds[i + 1: num_folds])

        # The i-th element is chosen to be the validation set
        X_validation_set = X_train_folds[i]
        y_validation_set = y_train_folds[i]

        classifier = KNearestNeighbor()
        classifier.train(X_train_set, y_train_set)
        y_validation_pred = classifier.predict(X_validation_set, k = k)
```

```
        # num_test = X_validation_set.shape[0]
        # num_correct = np.sum(y_validation_pred == y_validation_set)
        # accuracy = float(num_correct) / float(num_test)

        # The concise way to calculate accuracy
        accuracy = np.mean(y_validation_pred == y_validation_set)

        k_to_accuracies[k].append(accuracy)


# *****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 [16]:

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

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

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

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

Inline Question 3

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

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

YourAnswer: 2,4

YourExplanation:

1. False. When k is not 1 the decision boundary can be curved instead of linear.
2. True. For 1-NN, the training error is always 0 since the data point has been used to train the model and the one nearest neighbor is just itself. But the training error is no less than 0 for 5-NN.
3. False. From the cross-validation plot above we can see the test result of 5-NN can be more accurate than that of 1-NN.
4. True. Because in order to classify a test sample we need to explore the entire data set and sort the difference. When the size of training set gets larger, more time will be needed.

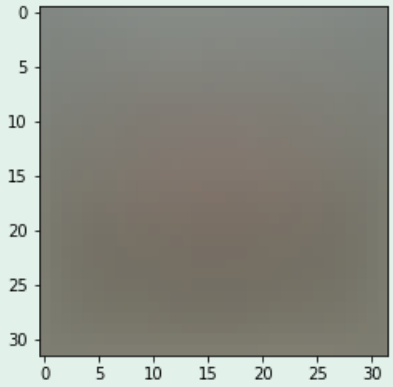
Multiclass Support Vector Machine exercise

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

```
[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)
```

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 `compute_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

In [7]:

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

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

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

```
loss: 8.855219
```

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 [12]:

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

```
# 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: -5.614316 analytic: -5.614316, relative error: 2.136001e-11
numerical: 3.915729 analytic: 3.915729, relative error: 8.582426e-13
numerical: -9.291898 analytic: -9.291898, relative error: 9.006908e-12
numerical: 19.063197 analytic: 19.063197, relative error: 1.117650e-11
numerical: -34.205150 analytic: -34.205150, relative error: 4.069945e-12
numerical: -30.054175 analytic: -29.979092, relative error: 1.250697e-03
numerical: -5.614316 analytic: -5.614316, relative error: 2.136001e-11
numerical: -14.295408 analytic: -14.274346, relative error: 7.372315e-04
numerical: 17.122491 analytic: 17.122491, relative error: 2.221451e-11
numerical: 1.135152 analytic: 1.135152, relative error: 5.424631e-10
numerical: -9.357989 analytic: -9.357989, relative error: 2.026877e-11
numerical: 4.358376 analytic: 4.358376, relative error: 6.973882e-12
numerical: 6.601013 analytic: 6.601013, relative error: 8.083296e-11
numerical: 4.024407 analytic: 4.024407, relative error: 1.108147e-11
numerical: -0.918634 analytic: -0.918634, relative error: 2.291302e-10
numerical: -6.571149 analytic: -6.571149, relative error: 2.137595e-11
numerical: -50.784805 analytic: -50.784805, relative error: 4.555410e-12
numerical: 5.850735 analytic: 5.850735, relative error: 5.973168e-11
numerical: 23.451292 analytic: 23.451292, relative error: 1.463956e-12
numerical: -38.818530 analytic: -38.813444, relative error: 6.550628e-05
```

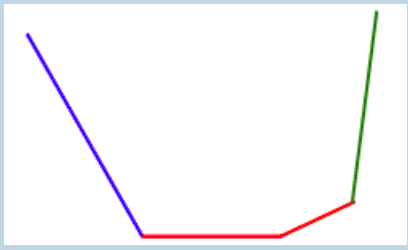
Inline Question 1

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*

YourAnswer: *fill this in.*

Yes, it is possible that a dimension in the gradcheck will not match exactly.

Such discrepancy may be caused by non-differentiable loss functions.



For example, in the image above (copied from lecture 3 optimization note), the kinks in the loss function (due to the max operation) technically make the loss function non-differentiable because at these kinks the gradient is not defined. At that time, numerical solution and analytical solution will be different.

In [13]:

```
# 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
```

```
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: 8.855219e+00 computed in 0.082196s
Vectorized loss: 8.855219e+00 computed in 0.008733s
difference: 0.000000
```

In [14]:

```
# 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 0.083400s
Vectorized loss and gradient: computed in 0.009060s
difference: 0.000000
```

Stochastic Gradient Descent

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

In [15]:

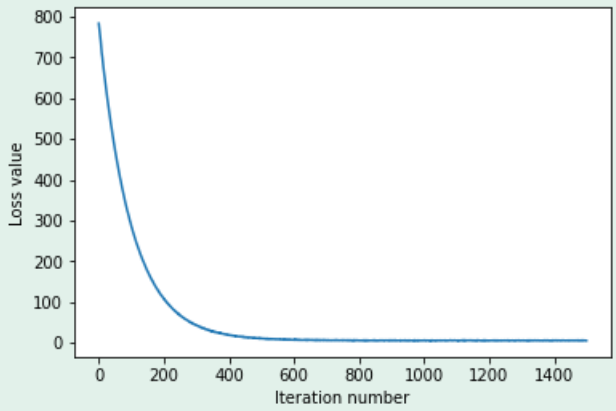
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                  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

```
iteration 0 / 1500: loss 783.192170
iteration 100 / 1500: loss 286.241058
iteration 200 / 1500: loss 107.962695
iteration 300 / 1500: loss 42.450987
iteration 400 / 1500: loss 19.047796
iteration 500 / 1500: loss 10.081456
iteration 600 / 1500: loss 7.199786
iteration 700 / 1500: loss 5.856792
iteration 800 / 1500: loss 5.269401
iteration 900 / 1500: loss 5.723489
iteration 1000 / 1500: loss 5.822409
iteration 1100 / 1500: loss 5.038547
iteration 1200 / 1500: loss 4.876252
iteration 1300 / 1500: loss 5.165719
iteration 1400 / 1500: loss 5.148642
That took 5.628599s
```

In [16]:

```
# A useful debugging strategy is to plot the loss as a function of
```

```
# 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 [17]:

```
# 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), ))
```

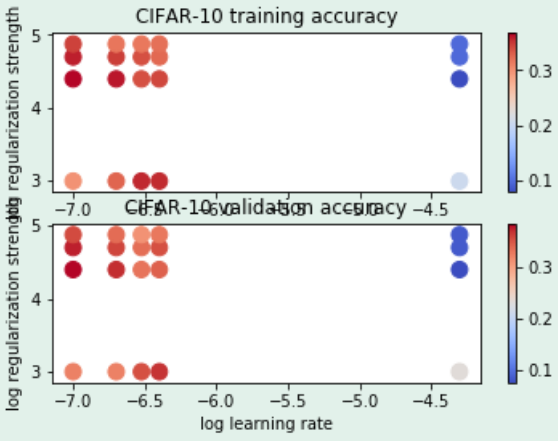
```
training accuracy: 0.368755
validation accuracy: 0.375000
```

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 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.

learning_rates = [1e-7, 2e-7, 3e-7, 4e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4, 7.5e4, 1e3]

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

for rate in learning_rates:
    for strength in regularization_strengths:
        # Initialize a new svm every time, and train the data
        svm = LinearSVM()
        svm.train(X_train, y_train, rate, strength, num_iters=1000)
```

```
        y_train_pred = svm.predict(X_train)
        training_accuracy = np.mean(y_train == y_train_pred)

        y_val_pred = svm.predict(X_val)
        validation_accuracy = np.mean(y_val == y_val_pred)

        # Store every result
        results[(rate, strength)] = (training_accuracy, validation_accuracy)

        # Store the best validation accuracy as the best_val
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_svm = svm

# *****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)
```

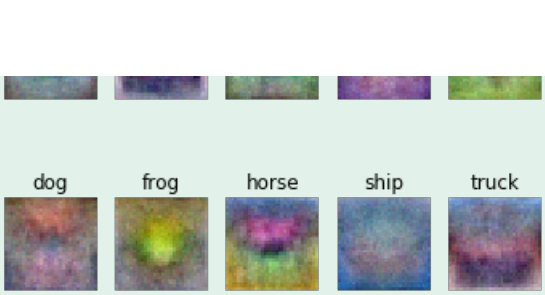
```
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.300347 val accuracy: 0.324000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.367143 val accuracy: 0.383000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.359837 val accuracy: 0.375000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.346327 val accuracy: 0.356000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.328469 val accuracy: 0.324000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.362571 val accuracy: 0.371000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.348571 val accuracy: 0.360000
lr 2.000000e-07 reg 7.500000e+04 train accuracy: 0.318592 val accuracy: 0.337000
lr 3.000000e-07 reg 1.000000e+03 train accuracy: 0.357878 val accuracy: 0.354000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.340163 val accuracy: 0.332000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.339449 val accuracy: 0.336000
lr 3.000000e-07 reg 7.500000e+04 train accuracy: 0.317531 val accuracy: 0.312000
lr 4.000000e-07 reg 1.000000e+03 train accuracy: 0.356673 val accuracy: 0.368000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.344143 val accuracy: 0.344000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.326082 val accuracy: 0.354000
lr 4.000000e-07 reg 7.500000e+04 train accuracy: 0.321306 val accuracy: 0.330000
lr 5.000000e-05 reg 1.000000e+03 train accuracy: 0.206163 val accuracy: 0.234000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.080163 val accuracy: 0.077000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 7.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.383000
```



In [21]:

```
# 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: 0.370000
```



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.

YourAnswer: *fill this in*

The visualized SVM weights look like to have rough outlines of the corresponding objects but they are really unclear.

This is because the weights should present the basic features of a certain type of item but the same kind of items may have really different shapes and colors. So the SVM weight should also be relatively blurry in order to resemble different items that belong to the same class.

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

Softmax Classifier

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

In [4]:

```
# 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: 2.418624
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer: *Fill this in*

Because in this case W is a (3073, 10) matrix. There are 10 classes, so the exponential of the correct score divided by the sum of the exponentials of all scores of 10 classes is approximately 0.1 and the final loss should be close to -log(0.1)

In [5]:

```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from 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: -1.479105 analytic: -1.479105, relative error: 4.890182e-09
numerical: 1.586017 analytic: 1.586017, relative error: 2.929528e-08
```

```
numerical: 1.905107 analytic: 1.905107, relative error: 2.770694e-08
numerical: -1.349112 analytic: -1.349113, relative error: 2.421673e-08
numerical: -0.049770 analytic: -0.049770, relative error: 2.810320e-07
numerical: -0.658285 analytic: -0.658285, relative error: 1.967038e-09
numerical: 4.038681 analytic: 4.038681, relative error: 1.313990e-08
numerical: -2.009209 analytic: -2.009209, relative error: 1.187446e-08
numerical: 0.855620 analytic: 0.855620, relative error: 1.601767e-08
numerical: 0.301801 analytic: 0.301801, relative error: 6.025370e-08
numerical: 3.564181 analytic: 3.564181, relative error: 1.145265e-08
numerical: -0.006918 analytic: -0.006918, relative error: 1.383980e-06
numerical: 1.061391 analytic: 1.061391, relative error: 3.158600e-08
numerical: 2.230215 analytic: 2.230215, relative error: 1.051158e-08
numerical: -0.047113 analytic: -0.047113, relative error: 3.201433e-07
numerical: -0.803367 analytic: -0.803367, relative error: 1.867133e-09
numerical: -0.554875 analytic: -0.554875, relative error: 4.480239e-08
numerical: 2.945169 analytic: 2.945169, relative error: 2.223602e-08
numerical: -2.765114 analytic: -2.765114, relative error: 8.125281e-09
numerical: 2.225074 analytic: 2.225074, relative error: 1.806905e-09
```

In [6]:

```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
naive loss: 2.418624e+00 computed in 0.110465s
vectorized loss: 2.418624e+00 computed in 0.025973s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [10]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.
from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 2e-7, 3e-7, 4e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4, 7.5e4, 1e3]

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

for rate in learning_rates:
    for strength in regularization_strengths:
        # Initialize a new svm every time, and train the data
        sftm = Softmax()
        sftm.train(X_train, y_train, learning_rate=rate, reg=strength, num_iters=5000)

        y_train_pred = sftm.predict(X_train)
        training_accuracy = np.mean(y_train == y_train_pred)
```

```
        y_val_pred = sftm.predict(X_val)
        validation_accuracy = np.mean(y_val == y_val_pred)

        # Store every result
        results[(rate, strength)] = (training_accuracy, validation_accuracy)

        # Store the best validation accuracy as the best_val
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_softmax = sftm

# *****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 1.000000e-07 reg 1.000000e+03 train accuracy: 0.363245 val accuracy: 0.364000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.327857 val accuracy: 0.345000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.303612 val accuracy: 0.320000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.297469 val accuracy: 0.310000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.400714 val accuracy: 0.399000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.329388 val accuracy: 0.344000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.303204 val accuracy: 0.320000
lr 2.000000e-07 reg 7.500000e+04 train accuracy: 0.276735 val accuracy: 0.287000
lr 3.000000e-07 reg 1.000000e+03 train accuracy: 0.406000 val accuracy: 0.408000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.331082 val accuracy: 0.338000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.310776 val accuracy: 0.324000
lr 3.000000e-07 reg 7.500000e+04 train accuracy: 0.305102 val accuracy: 0.310000
lr 4.000000e-07 reg 1.000000e+03 train accuracy: 0.403735 val accuracy: 0.404000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.325224 val accuracy: 0.345000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.311224 val accuracy: 0.321000
lr 4.000000e-07 reg 7.500000e+04 train accuracy: 0.292061 val accuracy: 0.301000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.406204 val accuracy: 0.409000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.322837 val accuracy: 0.338000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.298327 val accuracy: 0.314000
lr 5.000000e-07 reg 7.500000e+04 train accuracy: 0.297367 val accuracy: 0.311000
best validation accuracy achieved during cross-validation: 0.409000
```

In [11]:

```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.392000
```

Inline Question 2 - *True or False*

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

YourAnswer: True

YourExplanation: In the SVM, if the score of the new data point is out of the margin range from the correct class score, the loss wouldn't change. But in the Softmax loss, since the calculation formula of loss involves all scores, when a new datapoint is added the loss of Softmax will definitely change.

In [12]:

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

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for i in range(10):
```

```
    plt.subplot(2, 5, i + 1)

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

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.

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:
3.6802720496109664e-08
```

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
```

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 [5]:

```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

```
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
```

Train the network

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

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

In [6]:

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

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

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

```
Final training loss:  0.017149607938732093
```

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.

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 [8]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

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

Debug the training

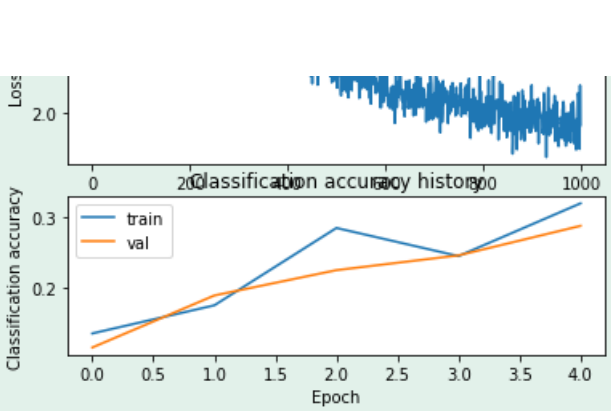
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

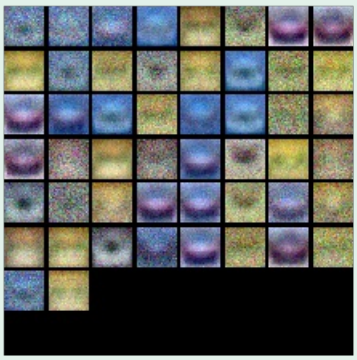
In [9]:

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

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



In [10]:

```
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)
```



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.

YourAnswer: I will try different combinations of hidden size, learning rate, regulation strength and batch size to find the appropriate hyperparameters.

In [13]:

```
best_net = None # store the best model into this
```

```
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)*****

best_acc = -1

input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = [50, 100]
learning_rates = [1e-3, 1e-4]
regulation_strengths = [0.25, 0.5]
batch_sizes = [200, 400]

for hidden_size in hidden_sizes:
    for rate in learning_rates:
        for strength in regulation_strengths:
            for batch_sz in batch_sizes:
                # Initialize a new network
                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=batch_sz,
                                  learning_rate=rate, learning_rate_decay=0.95,
                                  reg=strength, verbose=True)

                val_acc = (net.predict(X_val) == y_val).mean()

                print ('hidden size = %d, learning rate = %e, regulation strength = %e, batch_size
= %d, Valid_accuracy: %f'
                       %(hidden_size, rate, strength, batch_sz, val_acc))

                if val_acc > best_acc:
                    best_acc = val_acc
                    best_net = net




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

```
iteration 0 / 1000: loss 2.302938
iteration 100 / 1000: loss 2.001446
iteration 200 / 1000: loss 1.805014
iteration 300 / 1000: loss 1.692433
iteration 400 / 1000: loss 1.608849
iteration 500 / 1000: loss 1.619214
iteration 600 / 1000: loss 1.513565
iteration 700 / 1000: loss 1.816291
iteration 800 / 1000: loss 1.462572
iteration 900 / 1000: loss 1.499910
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size = 2
00, Valid_accuracy: 0.455000
iteration 0 / 1000: loss 2.302957
iteration 100 / 1000: loss 1.970983
iteration 200 / 1000: loss 1.760164
iteration 300 / 1000: loss 1.747722
iteration 400 / 1000: loss 1.611945
iteration 500 / 1000: loss 1.537459
iteration 600 / 1000: loss 1.506438
iteration 700 / 1000: loss 1.542277
iteration 800 / 1000: loss 1.529512
iteration 900 / 1000: loss 1.556933
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size = 4
00, Valid_accuracy: 0.483000
```

```
iteration 0 / 1000: loss 2.303354
iteration 100 / 1000: loss 1.884682
iteration 200 / 1000: loss 1.850019
iteration 300 / 1000: loss 1.746901
iteration 400 / 1000: loss 1.687540
iteration 500 / 1000: loss 1.803003
iteration 600 / 1000: loss 1.607671
iteration 700 / 1000: loss 1.563636
iteration 800 / 1000: loss 1.547632
iteration 900 / 1000: loss 1.558945
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size = 2
00, Valid_accuracy: 0.471000
iteration 0 / 1000: loss 2.303356
iteration 100 / 1000: loss 2.021878
iteration 200 / 1000: loss 1.782859
iteration 300 / 1000: loss 1.703777
iteration 400 / 1000: loss 1.655029
iteration 500 / 1000: loss 1.622439
iteration 600 / 1000: loss 1.661049
iteration 700 / 1000: loss 1.550663
iteration 800 / 1000: loss 1.557937
iteration 900 / 1000: loss 1.572061
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size = 4
00, Valid_accuracy: 0.468000
iteration 0 / 1000: loss 2.302958
iteration 100 / 1000: loss 2.302256
iteration 200 / 1000: loss 2.298498
iteration 300 / 1000: loss 2.260528
iteration 400 / 1000: loss 2.176528
iteration 500 / 1000: loss 2.163990
iteration 600 / 1000: loss 2.009059
iteration 700 / 1000: loss 2.048202
iteration 800 / 1000: loss 1.959462
iteration 900 / 1000: loss 2.070273
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size = 2
00, Valid_accuracy: 0.282000
iteration 0 / 1000: loss 2.302967
iteration 100 / 1000: loss 2.302552
iteration 200 / 1000: loss 2.298638
iteration 300 / 1000: loss 2.274630
iteration 400 / 1000: loss 2.197956
iteration 500 / 1000: loss 2.177668
iteration 600 / 1000: loss 2.053459
iteration 700 / 1000: loss 2.077726
iteration 800 / 1000: loss 2.005571
iteration 900 / 1000: loss 1.967835
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size = 4
00, Valid_accuracy: 0.283000
iteration 0 / 1000: loss 2.303389
iteration 100 / 1000: loss 2.303012
iteration 200 / 1000: loss 2.300792
iteration 300 / 1000: loss 2.279080
iteration 400 / 1000: loss 2.206246
iteration 500 / 1000: loss 2.102063
iteration 600 / 1000: loss 2.087562
iteration 700 / 1000: loss 2.007441
iteration 800 / 1000: loss 2.022864
iteration 900 / 1000: loss 2.090471
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size = 2
00, Valid_accuracy: 0.275000
iteration 0 / 1000: loss 2.303373
iteration 100 / 1000: loss 2.302982
iteration 200 / 1000: loss 2.298774
iteration 300 / 1000: loss 2.259631
iteration 400 / 1000: loss 2.216077
iteration 500 / 1000: loss 2.103669
iteration 600 / 1000: loss 2.107711
iteration 700 / 1000: loss 2.030601
iteration 800 / 1000: loss 2.016085
iteration 900 / 1000: loss 1.986063
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size = 4
00, Valid_accuracy: 0.280000
iteration 0 / 1000: loss 2.303338
iteration 100 / 1000: loss 1.911545
iteration 200 / 1000: loss 1.683111
iteration 300 / 1000: loss 1.666524
iteration 400 / 1000: loss 1.609624
```

```
iteration 500 / 1000: loss 1.615796
iteration 600 / 1000: loss 1.507984
iteration 700 / 1000: loss 1.513616
iteration 800 / 1000: loss 1.433492
iteration 900 / 1000: loss 1.593103
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size =
200, Valid_accuracy: 0.481000
iteration 0 / 1000: loss 2.303353
iteration 100 / 1000: loss 1.988648
iteration 200 / 1000: loss 1.750819
iteration 300 / 1000: loss 1.689663
iteration 400 / 1000: loss 1.568600
iteration 500 / 1000: loss 1.646333
iteration 600 / 1000: loss 1.453001
iteration 700 / 1000: loss 1.543148
iteration 800 / 1000: loss 1.474791
iteration 900 / 1000: loss 1.434732
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size =
400, Valid_accuracy: 0.493000
iteration 0 / 1000: loss 2.304104
iteration 100 / 1000: loss 1.992265
iteration 200 / 1000: loss 1.790595
iteration 300 / 1000: loss 1.683221
iteration 400 / 1000: loss 1.630720
iteration 500 / 1000: loss 1.694728
iteration 600 / 1000: loss 1.553691
iteration 700 / 1000: loss 1.627905
iteration 800 / 1000: loss 1.565421
iteration 900 / 1000: loss 1.665791
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size =
200, Valid_accuracy: 0.463000
iteration 0 / 1000: loss 2.304139
iteration 100 / 1000: loss 1.914435
iteration 200 / 1000: loss 1.812060
iteration 300 / 1000: loss 1.688253
iteration 400 / 1000: loss 1.691924
iteration 500 / 1000: loss 1.630675
iteration 600 / 1000: loss 1.515584
iteration 700 / 1000: loss 1.580681
iteration 800 / 1000: loss 1.561937
iteration 900 / 1000: loss 1.449985
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size =
400, Valid_accuracy: 0.487000
iteration 0 / 1000: loss 2.303320
iteration 100 / 1000: loss 2.301863
iteration 200 / 1000: loss 2.287207
iteration 300 / 1000: loss 2.234180
iteration 400 / 1000: loss 2.149615
iteration 500 / 1000: loss 2.142055
iteration 600 / 1000: loss 2.063547
iteration 700 / 1000: loss 2.073209
iteration 800 / 1000: loss 2.025929
iteration 900 / 1000: loss 2.002381
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size =
200, Valid_accuracy: 0.285000
iteration 0 / 1000: loss 2.303378
iteration 100 / 1000: loss 2.302502
iteration 200 / 1000: loss 2.292945
iteration 300 / 1000: loss 2.241410
iteration 400 / 1000: loss 2.161096
iteration 500 / 1000: loss 2.127070
iteration 600 / 1000: loss 2.055491
iteration 700 / 1000: loss 2.015337
iteration 800 / 1000: loss 2.030374
iteration 900 / 1000: loss 1.987144
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size =
400, Valid_accuracy: 0.291000
iteration 0 / 1000: loss 2.304117
iteration 100 / 1000: loss 2.303297
iteration 200 / 1000: loss 2.296294
iteration 300 / 1000: loss 2.257085
iteration 400 / 1000: loss 2.180464
iteration 500 / 1000: loss 2.089909
iteration 600 / 1000: loss 2.109783
iteration 700 / 1000: loss 2.042034
iteration 800 / 1000: loss 2.026012
iteration 900 / 1000: loss 1.958808
```

```
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size =
200, Valid_accuracy: 0.291000
iteration 0 / 1000: loss 2.304110
iteration 100 / 1000: loss 2.302891
iteration 200 / 1000: loss 2.294615
iteration 300 / 1000: loss 2.234861
iteration 400 / 1000: loss 2.171531
iteration 500 / 1000: loss 2.136451
iteration 600 / 1000: loss 2.038270
iteration 700 / 1000: loss 1.996320
iteration 800 / 1000: loss 2.046172
iteration 900 / 1000: loss 1.956266
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size =
400, Valid_accuracy: 0.297000
```

In [15]:

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

Run on the test set

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

In [17]:

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

```
Test accuracy:  0.481
```

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.

YourAnswer: 1

YourExplanation:

1. Correct. Training on a larger dataset can make the model become more general and decrease the gap.
2. Incorrect. The low testing accuracy may due to overfitting. Adding more hidden units may still lead to this problem.
3. Incorrect. From the process of tuning hyperparameters we can know that increasing the regulation strength may lead to lower accuracy.

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.

Train SVM on features

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

In [11]:

```
# 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]

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 rate in learning_rates:
    for strength in regularization_strengths:
        svm = LinearSVM()

        svm.train(X_train_feats, y_train, rate, strength, num_iters=1000)

        y_train_pred = svm.predict(X_train_feats)
        training_accuracy = np.mean(y_train == y_train_pred)

        y_val_pred = svm.predict(X_val_feats)
        validation_accuracy = np.mean(y_val == y_val_pred)

        # Store every result
        results[(rate, strength)] = (training_accuracy, validation_accuracy)

        # Store the best validation accuracy as the best_val
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_svm = svm

# *****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 1.000000e-09 reg 5.000000e+04 train accuracy: 0.102612 val accuracy: 0.105000
```

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.102612 val accuracy: 0.105000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.094776 val accuracy: 0.097000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.312796 val accuracy: 0.298000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.119388 val accuracy: 0.106000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.415592 val accuracy: 0.420000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.411531 val accuracy: 0.395000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415265 val accuracy: 0.425000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.404633 val accuracy: 0.406000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.302163 val accuracy: 0.333000
best validation accuracy achieved during cross-validation: 0.425000
```

In [12]:

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

```
0.419
```

In [13]:

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

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

Inline question 1:

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

YourAnswer: The misclassification results look similar in shape or background color. For example the first column is classified as plane, their shapes are like a straight stick and their background colors are mostly blue. This make sense because we capture the textures and colors of the pictures.

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 [21]:

```
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-2, 1e-1, 1]
regularization_strengths = [1e-3, 1e-2, 1e-1, 1]

results = {}
best_val = -1

for rate in learning_rates:
    for strength in regularization_strengths:
        net = TwoLayerNet(input_dim, hidden_dim, num_classes)

        net.train(X_train_feats, y_train, rate, strength, num_iters=1000)

        stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                          num_iters=1000, batch_size=200,
                          learning_rate=rate, learning_rate_decay=0.95,
                          reg= strength, verbose=True)

        y_train_pred = net.predict(X_train_feats)
        training_accuracy = np.mean(y_train == y_train_pred)

        y_val_pred = net.predict(X_val_feats)
        validation_accuracy = np.mean(y_val == y_val_pred)

        # Store every result
        results[(rate, strength)] = (training_accuracy, validation_accuracy)

        # Print out results.
        print('lr %e reg %e train accuracy: %f val accuracy: %f' % (rate, strength, train_accuracy,
val_accuracy))

        # Store the best validation accuracy as the best_val
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_net = net



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

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

```
iteration 0 / 1000: loss 2.302570
iteration 100 / 1000: loss 2.302482
iteration 200 / 1000: loss 2.302697
iteration 300 / 1000: loss 2.302547
iteration 400 / 1000: loss 2.302587
iteration 500 / 1000: loss 2.302597
iteration 600 / 1000: loss 2.302416
iteration 700 / 1000: loss 2.302114
iteration 800 / 1000: loss 2.302466
iteration 900 / 1000: loss 2.302180
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302624
iteration 100 / 1000: loss 2.302628
iteration 200 / 1000: loss 2.302626
```

```
iteration 300 / 1000: loss 2.302599
iteration 400 / 1000: loss 2.302298
iteration 500 / 1000: loss 2.302437
iteration 600 / 1000: loss 2.302588
iteration 700 / 1000: loss 2.301851
iteration 800 / 1000: loss 2.302178
iteration 900 / 1000: loss 2.302359
lr 1.000000e-02 reg 1.000000e-02 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302670
iteration 100 / 1000: loss 2.302473
iteration 200 / 1000: loss 2.302705
iteration 300 / 1000: loss 2.302314
iteration 400 / 1000: loss 2.302871
iteration 500 / 1000: loss 2.302476
iteration 600 / 1000: loss 2.302393
iteration 700 / 1000: loss 2.302601
iteration 800 / 1000: loss 2.302751
iteration 900 / 1000: loss 2.303035
lr 1.000000e-02 reg 1.000000e-01 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.303399
iteration 100 / 1000: loss 2.302626
iteration 200 / 1000: loss 2.302598
iteration 300 / 1000: loss 2.302792
iteration 400 / 1000: loss 2.302290
iteration 500 / 1000: loss 2.302472
iteration 600 / 1000: loss 2.302354
iteration 700 / 1000: loss 2.302781
iteration 800 / 1000: loss 2.302649
iteration 900 / 1000: loss 2.302364
lr 1.000000e-02 reg 1.000000e+00 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302591
iteration 100 / 1000: loss 2.301658
iteration 200 / 1000: loss 2.077883
iteration 300 / 1000: loss 1.808689
iteration 400 / 1000: loss 1.642119
iteration 500 / 1000: loss 1.607456
iteration 600 / 1000: loss 1.488302
iteration 700 / 1000: loss 1.468350
iteration 800 / 1000: loss 1.374784
iteration 900 / 1000: loss 1.462700
lr 1.000000e-01 reg 1.000000e-03 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302580
iteration 100 / 1000: loss 2.303193
iteration 200 / 1000: loss 2.162910
iteration 300 / 1000: loss 1.944073
iteration 400 / 1000: loss 1.788377
iteration 500 / 1000: loss 1.633414
iteration 600 / 1000: loss 1.644684
iteration 700 / 1000: loss 1.763350
iteration 800 / 1000: loss 1.609069
iteration 900 / 1000: loss 1.630226
lr 1.000000e-01 reg 1.000000e-02 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302674
iteration 100 / 1000: loss 2.303649
iteration 200 / 1000: loss 2.301822
iteration 300 / 1000: loss 2.293622
iteration 400 / 1000: loss 2.249799
iteration 500 / 1000: loss 2.262788
iteration 600 / 1000: loss 2.159708
iteration 700 / 1000: loss 2.196876
iteration 800 / 1000: loss 2.201736
iteration 900 / 1000: loss 2.195844
lr 1.000000e-01 reg 1.000000e-01 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.303426
iteration 100 / 1000: loss 2.302915
iteration 200 / 1000: loss 2.303354
iteration 300 / 1000: loss 2.302222
iteration 400 / 1000: loss 2.302448
iteration 500 / 1000: loss 2.302909
iteration 600 / 1000: loss 2.302180
iteration 700 / 1000: loss 2.303045
iteration 800 / 1000: loss 2.302386
iteration 900 / 1000: loss 2.303590
lr 1.000000e-01 reg 1.000000e+00 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302629
iteration 100 / 1000: loss 1.514732
iteration 200 / 1000: loss 1.166685
```

```
iteration 300 / 1000: loss 1.349503
iteration 400 / 1000: loss 1.486098
iteration 500 / 1000: loss 1.252753
iteration 600 / 1000: loss 1.268897
iteration 700 / 1000: loss 1.286091
iteration 800 / 1000: loss 1.286079
iteration 900 / 1000: loss 1.233952
lr 1.000000e+00 reg 1.000000e-03 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302666
iteration 100 / 1000: loss 1.643877
iteration 200 / 1000: loss 1.632431
iteration 300 / 1000: loss 1.671285
iteration 400 / 1000: loss 1.634135
iteration 500 / 1000: loss 1.628061
iteration 600 / 1000: loss 1.724921
iteration 700 / 1000: loss 1.647692
iteration 800 / 1000: loss 1.632797
iteration 900 / 1000: loss 1.553428
lr 1.000000e+00 reg 1.000000e-02 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.302642
iteration 100 / 1000: loss 2.161184
iteration 200 / 1000: loss 2.220938
iteration 300 / 1000: loss 2.244840
iteration 400 / 1000: loss 2.216159
iteration 500 / 1000: loss 2.191677
iteration 600 / 1000: loss 2.157572
iteration 700 / 1000: loss 2.144860
iteration 800 / 1000: loss 2.207613
iteration 900 / 1000: loss 2.137819
lr 1.000000e+00 reg 1.000000e-01 train accuracy: 0.100265 val accuracy: 0.087000
iteration 0 / 1000: loss 2.303367
iteration 100 / 1000: loss inf
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss inf
iteration 800 / 1000: loss inf
iteration 900 / 1000: loss nan
lr 1.000000e+00 reg 1.000000e+00 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved is: 0.577000
```

In [22]:

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

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

```
0.567
```

1 k_nearest_neighbor.py

```
from builtins import range
from builtins import object
import numpy as np
from past.builtins import xrange


class KNearestNeighbor(object):
    """ a kNN classifier with L2 distance """

    def __init__(self):
        pass

    def train(self, X, y):
        """
        Train the classifier. For k-nearest neighbors this is just
        memorizing the training data.

        Inputs:
        - X: A numpy array of shape (num_train, D) containing the training data
          consisting of num_train samples each of dimension D.
        - y: A numpy array of shape (N,) containing the training labels, where
             y[i] is the label for X[i].
        """
        self.X_train = X
        self.y_train = y

    def predict(self, X, k=1, num_loops=0):
        """
        Predict labels for test data using this classifier.

        Inputs:
        - X: A numpy array of shape (num_test, D) containing test data consisting
             of num_test samples each of dimension D.
        - k: The number of nearest neighbors that vote for the predicted labels.
        - num_loops: Determines which implementation to use to compute distances
          between training points and testing points.

        Returns:
        - y: A numpy array of shape (num_test,) containing predicted labels for the
          test data, where y[i] is the predicted label for the test point X[i].
        """
        if num_loops == 0:
            dists = self.compute_distances_no_loops(X)
        elif num_loops == 1:
            dists = self.compute_distances_one_loop(X)
        elif num_loops == 2:
            dists = self.compute_distances_two_loops(X)
        else:
            raise ValueError('Invalid value %d for num_loops' % num_loops)

        return self.predict_labels(dists, k=k)

    def compute_distances_two_loops(self, X):
        """
        Compute the distance between each test point in X and each training point
        in self.X_train using a nested loop over both the training data and the
        test data.

        Inputs:
        - X: A numpy array of shape (num_test, D) containing test data.

        Returns:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          is the Euclidean distance between the ith test point and the jth training
          point.
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in range(num_test):
            for j in range(num_train):
                #####################################################################
                # TODO:                                                             #
```

```
                # Compute the l2 distance between the ith test point and the jth    #
                # training point, and store the result in dists[i, j]. You should   #
                # not use a loop over dimension, nor use np.linalg.norm().          #
                #####################################################################
                # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

                dists[i, j] = np.sqrt(np.sum(np.square(X[i] - self.X_train[j])))

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

    def compute_distances_one_loop(self, X):
        """
        Compute the distance between each test point in X and each training point
        in self.X_train using a single loop over the test data.

        Input / Output: Same as compute_distances_two_loops
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in range(num_test):
            #######################################################################
            # TODO:                                                               #
            # Compute the l2 distance between the ith test point and all training #
            # points, and store the result in dists[i, :].                        #
            # Do not use np.linalg.norm().                                        #
            #######################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            dists[i] = np.sqrt(np.sum(np.square(X[i] - self.X_train), axis = 1))

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

    def compute_distances_no_loops(self, X):
        """
        Compute the distance between each test point in X and each training point
        in self.X_train using no explicit loops.

        Input / Output: Same as compute_distances_two_loops
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        #########################################################################
        # TODO:                                                                 #
        # Compute the l2 distance between all test points and all training      #
        # points without using any explicit loops, and store the result in      #
        # dists.                                                                #
        #                                                                       #
        # You should implement this function using only basic array operations; #
        # in particular you should not use functions from scipy,                #
        # nor use np.linalg.norm().                                             #
        #                                                                       #
        # HINT: Try to formulate the l2 distance using matrix multiplication    #
        #       and two broadcast sums.                                         #
        #########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        testSumOfSquare = np.sum(np.square(X), axis=1)
        # Get a (500,) matrix
        trainSumOfSquare = np.sum(np.square(self.X_train), axis=1)
        # Get a (5000,) matrix
        product = np.dot(X, self.X_train.T)
        # Get a (500, 5000) matrix
        dists = np.sqrt(testSumOfSquare[:, np.newaxis] + trainSumOfSquare - 2*product)
        # Add a new axis to testSumOfSquare to create a (500, 1) matrix and plus a (5000,) matrix to get a
    (500, 5000) matrix

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

    def predict_labels(self, dists, k=1):
        """
        Given a matrix of distances between test points and training points,
```

```
        predict a label for each test point.

        Inputs:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          gives the distance betwen the ith test point and the jth training point.

        Returns:
        - y: A numpy array of shape (num_test,) containing predicted labels for the
          test data, where y[i] is the predicted label for the test point X[i].
        """
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in range(num_test):
            # A list of length k storing the labels of the k nearest neighbors to
            # the ith test point.
            closest_y = []
            #########################################################################
            # TODO:                                                                 #
            # Use the distance matrix to find the k nearest neighbors of the ith    #
            # testing point, and use self.y_train to find the labels of these       #
            # neighbors. Store these labels in closest_y.                           #
            # Hint: Look up the function numpy.argsort.                             #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            index_ascending = np.argsort(dists[i])
            closest_y = self.y_train[index_ascending[:k]]


            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            #########################################################################
            # TODO:                                                                 #
            # Now that you have found the labels of the k nearest neighbors, you    #
            # need to find the most common label in the list closest_y of labels.   #
            # Store this label in y_pred[i]. Break ties by choosing the smaller     #
            # label.                                                                #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            y_pred[i] = np.argmax(np.bincount(closest_y))

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

        return y_pred
```

2 linear_classifier.py

```
from __future__ import print_function

from builtins import range
from builtins import object
import numpy as np
from cs231n.classifiers.linear_svm import *
from cs231n.classifiers.softmax import *
from past.builtins import xrange


class LinearClassifier(object):

    def __init__(self):
        self.W = None

    def train(self, X, y, learning_rate=1e-3, reg=1e-5, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this linear classifier using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) containing training data; there are N
          training samples each of dimension D.
        - y: A numpy array of shape (N,) containing training labels; y[i] = c
          means that X[i] has label 0 <= c < C for C classes.
        - learning_rate: (float) learning rate for optimization.
        - reg: (float) regularization strength.
        - num_iters: (integer) number of steps to take when optimizing
        - batch_size: (integer) number of training examples to use at each step.
        - verbose: (boolean) If true, print progress during optimization.

        Outputs:
        A list containing the value of the loss function at each training iteration.
        """
        num_train, dim = X.shape
        num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
        if self.W is None:
            # lazily initialize W
            self.W = 0.001 * np.random.randn(dim, num_classes)

        # Run stochastic gradient descent to optimize W
        loss_history = []
        for it in range(num_iters):
            X_batch = None
            y_batch = None

            #########################################################################
            # TODO:                                                                 #
            # Sample batch_size elements from the training data and their           #
            # corresponding labels to use in this round of gradient descent.        #
            # Store the data in X_batch and their corresponding labels in           #
            # y_batch; after sampling X_batch should have shape (batch_size, dim)   #
            # and y_batch should have shape (batch_size,)                           #
            #                                                                       #
            # Hint: Use np.random.choice to generate indices. Sampling with         #
            # replacement is faster than sampling without replacement.              #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            indices_Of_Batch = np.random.choice(num_train, batch_size)
            X_batch = X[indices_Of_Batch]
            y_batch = y[indices_Of_Batch]

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

            # evaluate loss and gradient
            loss, grad = self.loss(X_batch, y_batch, reg)
            loss_history.append(loss)

            # perform parameter update
            #########################################################################
            # TODO:                                                                 #
            # Update the weights using the gradient and the learning rate.          #
```

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

            self.W -= learning_rate * grad

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

            if verbose and it % 100 == 0:
                print('iteration %d / %d: loss %f' % (it, num_iters, loss))

        return loss_history

    def predict(self, X):
        """
        Use the trained weights of this linear classifier to predict labels for
        data points.

        Inputs:
        - X: A numpy array of shape (N, D) containing training data; there are N
          training samples each of dimension D.

        Returns:
        - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
          array of length N, and each element is an integer giving the predicted
          class.
        """
        y_pred = np.zeros(X.shape[0])
        ###########################################################################
        # TODO:                                                                   #
        # Implement this method. Store the predicted labels in y_pred.            #
        ###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # Get the largest index of every row in X.dot(self.W)
        y_pred = np.argmax(X.dot(self.W), axis = 1)

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

    def loss(self, X_batch, y_batch, reg):
        """
        Compute the loss function and its derivative.
        Subclasses will override this.

        Inputs:
        - X_batch: A numpy array of shape (N, D) containing a minibatch of N
          data points; each point has dimension D.
        - y_batch: A numpy array of shape (N,) containing labels for the minibatch.
        - reg: (float) regularization strength.

        Returns: A tuple containing:
        - loss as a single float
        - gradient with respect to self.W; an array of the same shape as W
        """
        pass


class LinearSVM(LinearClassifier):
    """ A subclass that uses the Multiclass SVM loss function """

    def loss(self, X_batch, y_batch, reg):
        return svm_loss_vectorized(self.W, X_batch, y_batch, reg)


class Softmax(LinearClassifier):
    """ A subclass that uses the Softmax + Cross-entropy loss function """

    def loss(self, X_batch, y_batch, reg):
        return softmax_loss_vectorized(self.W, X_batch, y_batch, reg)
```

3 linear_svm.py

```python
from builtins import range
import numpy as np
from random import shuffle
from past.builtins import xrange

def svm_loss_naive(W, X, y, reg):
    """
    Structured SVM loss function, naive implementation (with loops).

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.
    - reg: (float) regularization strength

    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
    dW = np.zeros(W.shape) # initialize the gradient as zero

    # compute the loss and the gradient
    num_classes = W.shape[1]
    num_train = X.shape[0]
    loss = 0.0
    for i in range(num_train):
        scores = X[i].dot(W)
        correct_class_score = scores[y[i]]
        for j in range(num_classes):
            if j == y[i]:
                continue
            margin = scores[j] - correct_class_score + 1 # note delta = 1
            if margin > 0:
                loss += margin

                # Subtract X[i].T from the y[i]-th column of dW
                dW[:, y[i]] -= X[i].T

                # Add X[i].T to the j-th column of dW if j != y[i]
                dW[:, j] += X[i].T

    # Right now the loss is a sum over all training examples, but we want it
    # to be an average instead so we divide by num_train.
    loss /= num_train

    # Add regularization to the loss.
    loss += reg * np.sum(W * W)

    #############################################################################
    # TODO:                                                                     #
    # Compute the gradient of the loss function and store it dW.                #
    # Rather that first computing the loss and then computing the derivative,   #
    # it may be simpler to compute the derivative at the same time that the     #
    # loss is being computed. As a result you may need to modify some of the    #
    # code above to compute the gradient.                                       #
    #############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # Want dW to be an average, divide it by num_train
    dW /= num_train

    # Add regularization to the gradient, which is the derivative of loss with respect to W
    dW += 2 * reg * W

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

    return loss, dW


```

```
def svm_loss_vectorized(W, X, y, reg):
    """
    Structured SVM loss function, vectorized implementation.

    Inputs and outputs are the same as svm_loss_naive.
    """
    loss = 0.0
    dW = np.zeros(W.shape) # initialize the gradient as zero

    #############################################################################
    # TODO:                                                                     #
    # Implement a vectorized version of the structured SVM loss, storing the    #
    # result in loss.                                                           #
    #############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    num_classes = W.shape[1]
    num_train = X.shape[0]
    scores = X.dot(W)

    # Diagonal elements are correct scores. Get the correct scores for every class, it is a (N,) matrix.
    correct_scores = scores[np.arange(num_train), y]

    # Clone the correct_scores column num_classes times to get a (N, C) matrix.
    correct_scores_matrix = np.array([correct_scores,] * num_classes).transpose()

    # Use scores to subtract correct scores.
    margins = np.maximum(0, scores - correct_scores_matrix + 1)

    # Diagonal elements in margins should be 0.
    margins[np.arange(num_train), y] = 0

    loss = np.sum(margins)
    loss /= num_train
    loss += reg * np.sum(W * W)

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

    #############################################################################
    # TODO:                                                                     #
    # Implement a vectorized version of the gradient for the structured SVM     #
    # loss, storing the result in dW.                                           #
    #                                                                           #
    # Hint: Instead of computing the gradient from scratch, it may be easier    #
    # to reuse some of the intermediate values that you used to compute the     #
    # loss.                                                                     #
    #############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # Create a margins_for_dW which has the same shape as margins
    margins_for_dW = np.zeros(margins.shape)

    # If there is a positive value in margins, the corresponding element in margins_for_dW is 1.
    margins_for_dW[margins > 0] = 1

    # Count the number of positive values in margins
    sum_Of_row = np.sum(margins_for_dW, axis = 1)

    # The diagonal elements in margins_for_dW should be the opposite value of sum_Of_row.
    margins_for_dW[np.arange(num_train), y] = -sum_Of_row

    # Use X.T multiply sum_Of_row
    dW = X.T.dot(margins_for_dW)/num_train + 2*reg*W

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

    return loss, dW
```

4 softmax.py

```
from builtins import range
import numpy as np
from random import shuffle
from past.builtins import xrange

def softmax_loss_naive(W, X, y, reg):
    """
    Softmax loss function, naive implementation (with loops)

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.
    - reg: (float) regularization strength

    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)

    #############################################################################
    # TODO: Compute the softmax loss and its gradient using explicit loops.     #
    # Store the loss in loss and the gradient in dW. If you are not careful     #
    # here, it is easy to run into numeric instability. Don't forget the        #
    # regularization!                                                           #
    #############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    num_classes = W.shape[1]
    num_train = X.shape[0]
    for i in range(num_train):
        scores = X[i].dot(W)

        # In order to avoid numeric instability, make the highest score to be zero
        scores -= np.max(scores)

        scores_exp_sum = np.sum(np.exp(scores))
        correct_score_exp = np.exp(scores[y[i]])
        loss += -np.log(correct_score_exp / scores_exp_sum)

        # Calculate the gradient of W
        for j in range(num_classes):
            dW[:, j] += - ((j == y[i]) - (np.exp(scores[j])/ scores_exp_sum)) * X[i]

    loss /= num_train
    loss += reg * np.sum(W * W)
    dW /= num_train
    dW += 2 * reg * W



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

    return loss, dW


def softmax_loss_vectorized(W, X, y, reg):
    """
    Softmax loss function, vectorized version.

    Inputs and outputs are the same as softmax_loss_naive.
    """
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)

```

```
    #############################################################################
    # TODO: Compute the softmax loss and its gradient using no explicit loops.  #
    # Store the loss in loss and the gradient in dW. If you are not careful     #
    # here, it is easy to run into numeric instability. Don't forget the        #
    # regularization!                                                           #
    #############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    num_classes = W.shape[1]
    num_train = X.shape[0]
    scores = X.dot(W)

    # Get the maximum score of every row and clone it multiple times to be a (N, C) matrix
    max_scores = np.amax(scores, axis = 1)
    max_scores_matrix = np.array([max_scores,] * num_classes).transpose()
    scores -= max_scores_matrix

    correct_scores = scores[np.arange(num_train), y]
    scores_row_exp_sum = np.sum(np.exp(scores), axis = 1)
    loss = np.sum(-np.log(np.exp(correct_scores) / scores_row_exp_sum))

    loss /= num_train
    loss += reg * np.sum(W * W)


    # Calculate the gradient of W
    margins_for_dW = np.exp(scores) / scores_row_exp_sum.reshape(num_train, 1)
    # The diagonal elements should minus 1
    margins_for_dW[np.arange(num_train), y] -= 1
    dW = X.T.dot(margins_for_dW)

    dW /= num_train
    dW += 2 * reg * W









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

    return loss, dW
```

5 neural_net.py

```python
from __future__ import print_function

from builtins import range
from builtins import object
import numpy as np
import matplotlib.pyplot as plt
from past.builtins import xrange

class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network. The net has an input dimension of
    N, a hidden layer dimension of H, and performs classification over C classes.
    We train the network with a softmax loss function and L2 regularization on the
    weight matrices. The network uses a ReLU nonlinearity after the first fully
    connected layer.

    In other words, the network has the following architecture:

    input - fully connected layer - ReLU - fully connected layer - softmax

    The outputs of the second fully-connected layer are the scores for each class.
    """

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        """
        Initialize the model. Weights are initialized to small random values and
        biases are initialized to zero. Weights and biases are stored in the
        variable self.params, which is a dictionary with the following keys:

        W1: First layer weights; has shape (D, H)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (H, C)
        b2: Second layer biases; has shape (C,)

        Inputs:
        - input_size: The dimension D of the input data.
        - hidden_size: The number of neurons H in the hidden layer.
        - output_size: The number of classes C.
        """
        self.params = {}
        self.params['W1'] = std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural
        network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
          an integer in the range 0 <= y[i] < C. This parameter is optional; if it
          is not passed then we only return scores, and if it is passed then we
          instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
        the score for class c on input X[i].

        If y is not None, instead return a tuple of:
        - loss: Loss (data loss and regularization loss) for this batch of training
          samples.
        - grads: Dictionary mapping parameter names to gradients of those parameters
          with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

```

```
        # Compute the forward pass
        scores = None
        ###############################################################################
        # TODO: Perform the forward pass, computing the class scores for the input.   #
        # Store the result in the scores variable, which should be an array of        #
        # shape (N, C).                                                               #
        ###############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        h1 = np.maximum(0, np.dot(X, W1) + b1)
        h2 = np.dot(h1, W2) + b2
        scores = h2

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

        # If the targets are not given then jump out, we're done
        if y is None:
            return scores

        # Compute the loss
        loss = None
        ###############################################################################
        # TODO: Finish the forward pass, and compute the loss. This should include    #
        # both the data loss and L2 regularization for W1 and W2. Store the result    #
        # in the variable loss, which should be a scalar. Use the Softmax             #
        # classifier loss.                                                            #
        ###############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        num_train = X.shape[0]

        # Avoid numerical instability
        scores -= np.amax(scores, axis = 1).reshape(scores.shape[0], 1)

        scores_exp = np.exp(scores)
        scores_exp_sums = np.sum(scores_exp, axis=1)
        correct_scores_exp = scores_exp[range(num_train), y]
        loss = correct_scores_exp / scores_exp_sums

        loss = np.sum(-np.log(loss))
        loss /= num_train
        loss += reg * (np.sum(W1 * W1) + np.sum(W2 * W2))

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

        # Backward pass: compute gradients
        grads = {}
        ###############################################################################
        # TODO: Compute the backward pass, computing the derivatives of the weights   #
        # and biases. Store the results in the grads dictionary. For example,         #
        # grads['W1'] should store the gradient on W1, and be a matrix of same size   #
        ###############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        margins_for_dW2 = scores_exp / scores_exp_sums.reshape(num_train, 1)

        # The diagonal elements are correct and they should minus 1
        margins_for_dW2[np.arange(num_train), y] -= 1
        margins_for_dW2 /= num_train

        dW2 = h1.T.dot(margins_for_dW2)
        db2 = np.sum(margins_for_dW2, axis = 0)

        margins_for_hidden = margins_for_dW2.dot(W2.T)
        margins_for_hidden[h1 <= 0] = 0

        dW1 = X.T.dot(margins_for_hidden)
        db1 = np.sum(margins_for_hidden, axis = 0)

        dW2 += 2 * reg * W2
        dW1 += 2 * reg * W1

        grads['W1'] = dW1
        grads['b1'] = db1
        grads['W2'] = dW2
        grads['b2'] = db2
```

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

        return loss, grads

    def train(self, X, y, X_val, y_val,
              learning_rate=1e-3, learning_rate_decay=0.95,
              reg=5e-6, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this neural network using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) giving training data.
        - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
          X[i] has label c, where 0 <= c < C.
        - X_val: A numpy array of shape (N_val, D) giving validation data.
        - y_val: A numpy array of shape (N_val,) giving validation labels.
        - learning_rate: Scalar giving learning rate for optimization.
        - learning_rate_decay: Scalar giving factor used to decay the learning rate
          after each epoch.
        - reg: Scalar giving regularization strength.
        - num_iters: Number of steps to take when optimizing.
        - batch_size: Number of training examples to use per step.
        - verbose: boolean; if true print progress during optimization.
        """
        num_train = X.shape[0]
        iterations_per_epoch = max(num_train / batch_size, 1)

        # Use SGD to optimize the parameters in self.model
        loss_history = []
        train_acc_history = []
        val_acc_history = []

        for it in range(num_iters):
            X_batch = None
            y_batch = None

            #########################################################################
            # TODO: Create a random minibatch of training data and labels, storing  #
            # them in X_batch and y_batch respectively.                             #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            batch_indices = np.random.choice(num_train, batch_size)
            X_batch = X[batch_indices]
            y_batch = y[batch_indices]

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

            # Compute loss and gradients using the current minibatch
            loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
            loss_history.append(loss)

            #########################################################################
            # TODO: Use the gradients in the grads dictionary to update the         #
            # parameters of the network (stored in the dictionary self.params)      #
            # using stochastic gradient descent. You'll need to use the gradients   #
            # stored in the grads dictionary defined above.                         #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            self.params['W1'] += -grads['W1']*learning_rate
            self.params['b1'] += -grads['b1']*learning_rate
            self.params['W2'] += -grads['W2']*learning_rate
            self.params['b2'] += -grads['b2']*learning_rate

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

            if verbose and it % 100 == 0:
                print('iteration %d / %d: loss %f' % (it, num_iters, loss))

            # Every epoch, check train and val accuracy and decay learning rate.
            if it % iterations_per_epoch == 0:
                # Check accuracy
                train_acc = (self.predict(X_batch) == y_batch).mean()
```

```
                val_acc = (self.predict(X_val) == y_val).mean()
                train_acc_history.append(train_acc)
                val_acc_history.append(val_acc)

                # Decay learning rate
                learning_rate *= learning_rate_decay

        return {
          'loss_history': loss_history,
          'train_acc_history': train_acc_history,
          'val_acc_history': val_acc_history,
        }

    def predict(self, X):
        """
        Use the trained weights of this two-layer network to predict labels for
        data points. For each data point we predict scores for each of the C
        classes, and assign each data point to the class with the highest score.

        Inputs:
        - X: A numpy array of shape (N, D) giving N D-dimensional data points to
          classify.

        Returns:
        - y_pred: A numpy array of shape (N,) giving predicted labels for each of
          the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
          to have class c, where 0 <= c < C.
        """
        y_pred = None

        ###########################################################################
        # TODO: Implement this function; it should be VERY simple!                #
        ###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        h1 = np.maximum(0, np.dot(X, W1) + b1)
        h2 = np.dot(h1, W2) + b2
        scores = h2
        y_pred = np.argmax(scores, axis = 1)

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

        return y_pred
```