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.

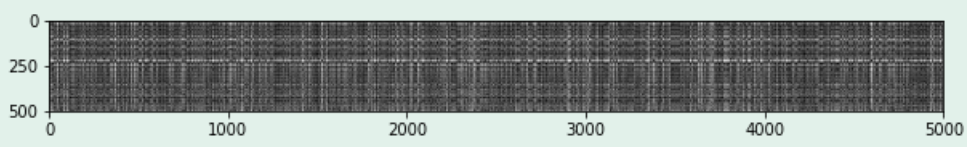
In [8]:

```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

In [9]:

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



Inline Question 1

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

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

YourAnswer: Bright rows are indicative of a test photo being largely different from a majority of the training examples, resulting in high distance along the entire row corresponding to that test image. The columns indicate a particular training image which has high distance from most training images, signifying a lack of similar pixel values in the test set.

In [10]:

```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

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

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

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

In [11]:

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

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

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

Inline Question 2

We can also 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 3

YourExplanation:

Subtracting a uniform amount from every pixel will not affect the difference between pixels, and dividing them all by a uniform amount will scale the distance, but since this scale will affect all of the pixels the same way, the distances will not scale differently relative to each other.

Pixel-wise means and standard deviations will have different effects on each pixel-wise L1 difference. This could affect some of the outlier pixel differences, changing the resulting predictions.

```
One loop difference was: 0.000000
```

```
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 39.045972 seconds
One loop version took 46.012859 seconds
No loop version took 0.265476 seconds
```

Cross-validation

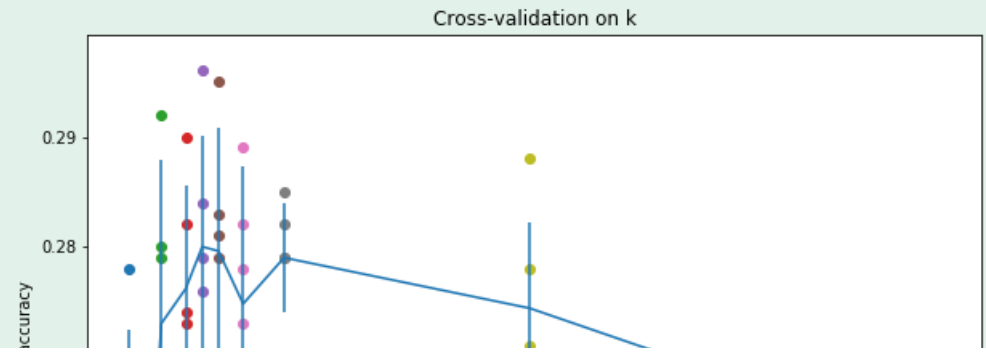
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

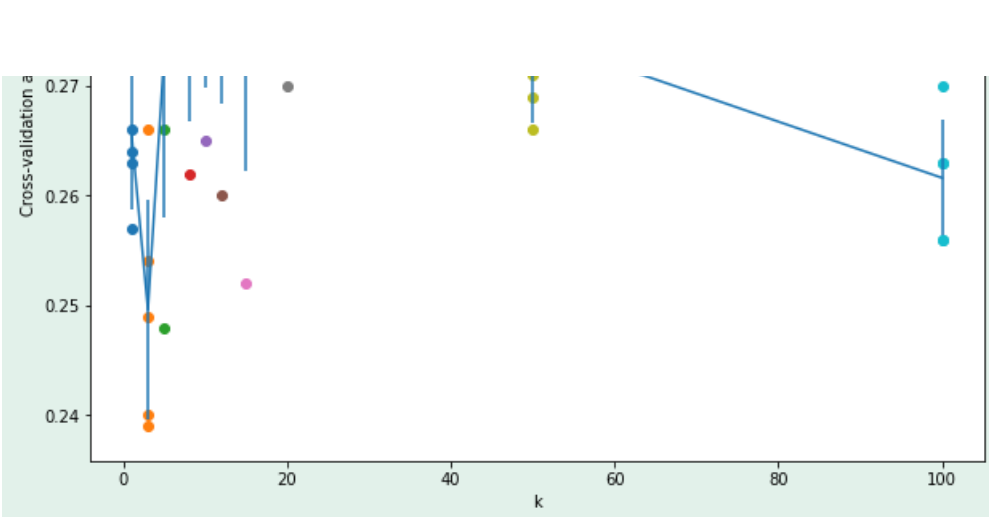
In [50]:

```
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.split(X_train, num_folds, axis=0)
Y_train_folds = np.split(y_train, num_folds, axis=0)

# *****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 ki in k_choices:
    for v in range(num_folds):
        #build training and validation sets
        Xt = np.empty([1,X_train_folds[0].shape[1]])
        Yt = np.empty([1,1])
        Xv = X_train_folds[v]
        Yv = Y_train_folds[v]
        for f in range(num_folds):
            if f!=v:
                Xt = np.append(Xt, X_train_folds[f], axis=0)
                Yt = np.append(Yt, Y_train_folds[f])

        #train on sets
        classifier.train(Xt, Yt)

        #compute dists
        dst = classifier.compute_distances_no_loops(Xv)

        #predict
        Yp = classifier.predict_labels(dst, k=ki)
        num_correct = np.sum(Yp == Yv)
        accuracy = float(num_correct) / Xv.shape[0]
```

```
        if ki in k_to_accuracies:
            k_to_accuracies[ki] = np.append(k_to_accuracies[ki], accuracy)
        else:
            k_to_accuracies[ki] = 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 = 0.279000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.274000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.279000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.281000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.273000
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 [58]:

```
# 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:

2- A 1-NN will simply make a 1:1 match from training images it is classifying to that same training image in its memory, and correctly classify, giving it perfect training accuracy. A 5-NN may miss out on properly classifying outlier training images, although this will probably help it generalize better than the 1-NN. 4- The k-NN classifier must compare the input images to every single image in the trainnig set before reaching a decision.

In [ ]:

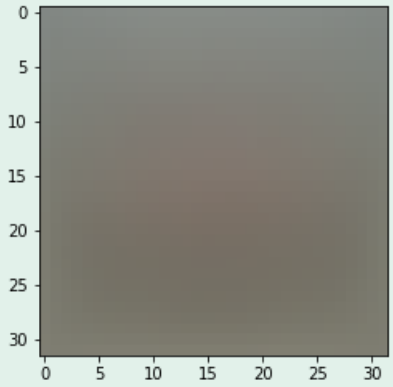
Multiclass Support Vector Machine exercise

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: 8.357117
```

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

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

In [8]:

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

```
# 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: -14.504248 analytic: -14.504248, relative error: 5.040314e-13
numerical: 8.238786 analytic: 8.238786, relative error: 5.245035e-11
numerical: -11.748661 analytic: -11.748661, relative error: 1.500822e-11
numerical: 31.945226 analytic: 31.945226, relative error: 7.074627e-12
numerical: -2.133373 analytic: -2.133373, relative error: 1.054957e-10
numerical: 14.246935 analytic: 14.246935, relative error: 4.743581e-13
numerical: 1.362021 analytic: 1.362021, relative error: 9.427860e-11
numerical: -33.539966 analytic: -33.539966, relative error: 4.495342e-12
numerical: 6.755003 analytic: 6.755003, relative error: 9.875145e-12
numerical: -12.820682 analytic: -12.820682, relative error: 4.037256e-12
numerical: -0.948241 analytic: -0.948241, relative error: 6.376765e-11
numerical: -9.637857 analytic: -9.637857, relative error: 2.827275e-11
numerical: 28.009333 analytic: 28.009333, relative error: 2.262388e-12
numerical: -25.548358 analytic: -25.548358, relative error: 1.606674e-11
numerical: -21.837787 analytic: -21.837787, relative error: 1.461429e-11
numerical: -29.515984 analytic: -29.515984, relative error: 1.118800e-12
numerical: 40.062817 analytic: 40.062817, relative error: 1.110078e-12
numerical: -6.899935 analytic: -6.899935, relative error: 5.852301e-11
numerical: -15.736079 analytic: -15.736079, relative error: 3.146538e-12
numerical: 15.027298 analytic: 15.027298, relative error: 6.559812e-12
```

Inline Question 1

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

YourAnswer:

This discrepancy is likely caused by the hinge having nondifferentiable points, in 1D an example is f(x) = max(0,x), near x=0 the analytical gradient will have an undefined point but the numerical gradient will approximate some change over the small region x=[-0.001,0.001] where the change in y is something slightly less than the change in x. The larger the margin is, the more likely this is to happen. This shouldn't be a very concerning situation as the gradient in these cases is still indicative of thw loss function generally increasing or decreasing in that direction.

In [9]:

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

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

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

```
Naive loss: 8.357117e+00 computed in 0.112699s
(500, 10)
Vectorized loss: 8.357117e+00 computed in 0.003995s
```

```
difference: 0.000000
```

In [10]:

```
# 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()
nloss , 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()
vloss , 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 0.109707s
(500, 10)
Vectorized loss and gradient: computed in 0.003000s
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.

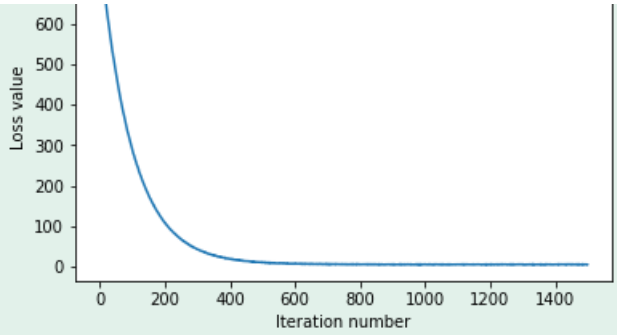
In [12]:

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

```
iteration 0 / 1500: loss 788.259629
iteration 100 / 1500: loss 289.069757
iteration 200 / 1500: loss 108.669248
iteration 300 / 1500: loss 42.623557
iteration 400 / 1500: loss 19.569938
iteration 500 / 1500: loss 10.618952
iteration 600 / 1500: loss 6.809590
iteration 700 / 1500: loss 5.778346
iteration 800 / 1500: loss 5.917785
iteration 900 / 1500: loss 5.072217
iteration 1000 / 1500: loss 5.892808
iteration 1100 / 1500: loss 5.442712
iteration 1200 / 1500: loss 5.532513
iteration 1300 / 1500: loss 4.982116
iteration 1400 / 1500: loss 5.507915
That took 8.525203s
```

In [13]:

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



In [14]:

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

```
training accuracy: 0.370204
validation accuracy: 0.384000
```

In [15]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 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, 5e-5]
regularization_strengths = [2.5e4, 5e4]

# 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 lr in np.logspace(-7,-4,10):
    for rs in np.arange(regularization_strengths[0],regularization_strengths[1],np.diff(regularizat
ion_strengths)/10):
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,
                              num_iters=1000, verbose=True)

        Yt = svm.predict(X_train)
        Yp = svm.predict(X_val)

        t_num_correct = np.sum(Yt == y_train)
        t_accuracy = float(t_num_correct) / X_train.shape[0]

        v_num_correct = np.sum(Yp == y_val)
        v_accuracy = float(v_num_correct) / X_val.shape[0]
```

```
        results[(lr,rs)] = (t_accuracy ,v_accuracy)

        if (best_val<v_accuracy):
            best_val = v_accuracy
            best_svm = svm

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

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

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

```
iteration 0 / 1000: loss 787.959228
iteration 100 / 1000: loss 285.739146
iteration 200 / 1000: loss 106.858658
iteration 300 / 1000: loss 42.260097
iteration 400 / 1000: loss 18.777131
iteration 500 / 1000: loss 9.990901
iteration 600 / 1000: loss 6.996069
iteration 700 / 1000: loss 6.393997
iteration 800 / 1000: loss 5.847828
iteration 900 / 1000: loss 5.266571
iteration 0 / 1000: loss 883.209753
iteration 100 / 1000: loss 291.571127
iteration 200 / 1000: loss 99.766127
iteration 300 / 1000: loss 36.160276
iteration 400 / 1000: loss 15.663899
iteration 500 / 1000: loss 8.710549
iteration 600 / 1000: loss 6.289606
iteration 700 / 1000: loss 5.408744
iteration 800 / 1000: loss 5.299582
iteration 900 / 1000: loss 5.246534
iteration 0 / 1000: loss 948.843841
iteration 100 / 1000: loss 284.107190
iteration 200 / 1000: loss 88.864060
iteration 300 / 1000: loss 29.307723
iteration 400 / 1000: loss 12.972605
iteration 500 / 1000: loss 8.308069
iteration 600 / 1000: loss 6.876864
iteration 700 / 1000: loss 5.708711
iteration 800 / 1000: loss 5.568939
iteration 900 / 1000: loss 6.059558
iteration 0 / 1000: loss 1029.557059
iteration 100 / 1000: loss 278.086944
iteration 200 / 1000: loss 78.520172
iteration 300 / 1000: loss 25.330013
iteration 400 / 1000: loss 10.196908
iteration 500 / 1000: loss 6.775396
iteration 600 / 1000: loss 5.971307
iteration 700 / 1000: loss 5.622102
iteration 800 / 1000: loss 5.202388
iteration 900 / 1000: loss 5.281063
iteration 0 / 1000: loss 1096.972740
iteration 100 / 1000: loss 268.755767
iteration 200 / 1000: loss 69.608308
iteration 300 / 1000: loss 21.344369
iteration 400 / 1000: loss 9.504325
iteration 500 / 1000: loss 6.751897
iteration 600 / 1000: loss 5.721568
iteration 700 / 1000: loss 5.787830
iteration 800 / 1000: loss 5.701452
iteration 900 / 1000: loss 5.447646
iteration 0 / 1000: loss 1171.068310
iteration 100 / 1000: loss 260.047421
iteration 200 / 1000: loss 60.559845
iteration 300 / 1000: loss 18.295300
iteration 400 / 1000: loss 8.210832
iteration 500 / 1000: loss 6.517498
iteration 600 / 1000: loss 5.656110
iteration 700 / 1000: loss 5.187818
iteration 800 / 1000: loss 5.311388
```

```
iteration 900 / 1000: loss 4.737349
iteration 0 / 1000: loss 1247.158948
iteration 100 / 1000: loss 250.528316
iteration 200 / 1000: loss 54.152160
iteration 300 / 1000: loss 15.037893
iteration 400 / 1000: loss 7.767941
iteration 500 / 1000: loss 6.158706
iteration 600 / 1000: loss 5.000304
iteration 700 / 1000: loss 5.621682
iteration 800 / 1000: loss 5.858246
iteration 900 / 1000: loss 5.664003
iteration 0 / 1000: loss 1332.229008
iteration 100 / 1000: loss 243.541170
iteration 200 / 1000: loss 47.916137
iteration 300 / 1000: loss 13.463414
iteration 400 / 1000: loss 6.957108
iteration 500 / 1000: loss 5.822774
iteration 600 / 1000: loss 5.530207
iteration 700 / 1000: loss 5.859266
iteration 800 / 1000: loss 5.024029
iteration 900 / 1000: loss 5.166398
iteration 0 / 1000: loss 1416.283093
iteration 100 / 1000: loss 232.778428
iteration 200 / 1000: loss 42.535224
iteration 300 / 1000: loss 12.415307
iteration 400 / 1000: loss 6.412868
iteration 500 / 1000: loss 5.759599
iteration 600 / 1000: loss 5.490158
iteration 700 / 1000: loss 5.868167
iteration 800 / 1000: loss 6.097463
iteration 900 / 1000: loss 5.831738
iteration 0 / 1000: loss 1487.506627
iteration 100 / 1000: loss 222.922547
iteration 200 / 1000: loss 38.002806
iteration 300 / 1000: loss 10.212421
iteration 400 / 1000: loss 6.515552
iteration 500 / 1000: loss 5.463017
iteration 600 / 1000: loss 5.571340
iteration 700 / 1000: loss 5.835463
iteration 800 / 1000: loss 5.356069
iteration 900 / 1000: loss 5.838510
iteration 0 / 1000: loss 800.741071
iteration 100 / 1000: loss 94.218634
iteration 200 / 1000: loss 15.307642
iteration 300 / 1000: loss 6.570206
iteration 400 / 1000: loss 5.129732
iteration 500 / 1000: loss 5.323507
iteration 600 / 1000: loss 5.111913
iteration 700 / 1000: loss 5.269733
iteration 800 / 1000: loss 5.646049
iteration 900 / 1000: loss 5.047217
iteration 0 / 1000: loss 858.133044
iteration 100 / 1000: loss 81.502400
iteration 200 / 1000: loss 12.305212
iteration 300 / 1000: loss 5.963774
iteration 400 / 1000: loss 5.596164
iteration 500 / 1000: loss 5.170256
iteration 600 / 1000: loss 5.601675
iteration 700 / 1000: loss 5.520876
iteration 800 / 1000: loss 5.385412
iteration 900 / 1000: loss 5.588075
iteration 0 / 1000: loss 926.096581
iteration 100 / 1000: loss 72.193226
iteration 200 / 1000: loss 10.139283
iteration 300 / 1000: loss 5.821281
iteration 400 / 1000: loss 4.551738
iteration 500 / 1000: loss 5.862026
iteration 600 / 1000: loss 5.793883
iteration 700 / 1000: loss 5.518888
iteration 800 / 1000: loss 4.878370
iteration 900 / 1000: loss 5.796935
iteration 0 / 1000: loss 1016.571084
iteration 100 / 1000: loss 64.119801
iteration 200 / 1000: loss 9.052246
iteration 300 / 1000: loss 5.987927
iteration 400 / 1000: loss 5.526989
iteration 500 / 1000: loss 5.451835
```

```
iteration 600 / 1000: loss 4.916852
iteration 700 / 1000: loss 5.496479
iteration 800 / 1000: loss 4.908552
iteration 900 / 1000: loss 5.285651
iteration 0 / 1000: loss 1095.863975
iteration 100 / 1000: loss 56.305486
iteration 200 / 1000: loss 7.505061
iteration 300 / 1000: loss 5.731673
iteration 400 / 1000: loss 5.501487
iteration 500 / 1000: loss 5.524283
iteration 600 / 1000: loss 5.959802
iteration 700 / 1000: loss 5.451081
iteration 800 / 1000: loss 5.609636
iteration 900 / 1000: loss 5.168363
iteration 0 / 1000: loss 1177.546099
iteration 100 / 1000: loss 49.261408
iteration 200 / 1000: loss 7.112876
iteration 300 / 1000: loss 6.374595
iteration 400 / 1000: loss 5.235376
iteration 500 / 1000: loss 5.402552
iteration 600 / 1000: loss 5.990004
iteration 700 / 1000: loss 5.532881
iteration 800 / 1000: loss 5.649714
iteration 900 / 1000: loss 5.464740
iteration 0 / 1000: loss 1250.980742
iteration 100 / 1000: loss 42.757957
iteration 200 / 1000: loss 6.789831
iteration 300 / 1000: loss 5.714242
iteration 400 / 1000: loss 6.140434
iteration 500 / 1000: loss 6.014220
iteration 600 / 1000: loss 5.430151
iteration 700 / 1000: loss 5.898554
iteration 800 / 1000: loss 5.378529
iteration 900 / 1000: loss 6.094934
iteration 0 / 1000: loss 1332.471804
iteration 100 / 1000: loss 38.128508
iteration 200 / 1000: loss 6.119339
iteration 300 / 1000: loss 5.691613
iteration 400 / 1000: loss 5.572715
iteration 500 / 1000: loss 5.617876
iteration 600 / 1000: loss 5.559999
iteration 700 / 1000: loss 5.636734
iteration 800 / 1000: loss 5.716089
iteration 900 / 1000: loss 5.223048
iteration 0 / 1000: loss 1408.975436
iteration 100 / 1000: loss 32.634319
iteration 200 / 1000: loss 6.168610
iteration 300 / 1000: loss 5.537758
iteration 400 / 1000: loss 5.976350
iteration 500 / 1000: loss 6.468412
iteration 600 / 1000: loss 4.826804
iteration 700 / 1000: loss 5.364719
iteration 800 / 1000: loss 5.535901
iteration 900 / 1000: loss 5.644447
iteration 0 / 1000: loss 1494.014122
iteration 100 / 1000: loss 28.929492
iteration 200 / 1000: loss 6.048226
iteration 300 / 1000: loss 5.874087
iteration 400 / 1000: loss 5.493999
iteration 500 / 1000: loss 5.869896
iteration 600 / 1000: loss 5.386139
iteration 700 / 1000: loss 5.748582
iteration 800 / 1000: loss 5.762264
iteration 900 / 1000: loss 5.428832
iteration 0 / 1000: loss 790.213344
iteration 100 / 1000: loss 12.228324
iteration 200 / 1000: loss 5.357938
iteration 300 / 1000: loss 5.742807
iteration 400 / 1000: loss 5.463366
iteration 500 / 1000: loss 5.315781
iteration 600 / 1000: loss 5.423737
iteration 700 / 1000: loss 5.876392
iteration 800 / 1000: loss 5.547438
iteration 900 / 1000: loss 5.446978
iteration 0 / 1000: loss 867.820061
iteration 100 / 1000: loss 10.363552
iteration 200 / 1000: loss 6.194421
```

```
iteration 300 / 1000: loss 6.076603
iteration 400 / 1000: loss 5.753401
iteration 500 / 1000: loss 5.926425
iteration 600 / 1000: loss 5.065185
iteration 700 / 1000: loss 5.985275
iteration 800 / 1000: loss 5.803659
iteration 900 / 1000: loss 5.325466
iteration 0 / 1000: loss 929.711666
iteration 100 / 1000: loss 8.593700
iteration 200 / 1000: loss 5.369303
iteration 300 / 1000: loss 6.080574
iteration 400 / 1000: loss 5.101876
iteration 500 / 1000: loss 5.825804
iteration 600 / 1000: loss 5.415620
iteration 700 / 1000: loss 5.607840
iteration 800 / 1000: loss 6.027271
iteration 900 / 1000: loss 6.656960
iteration 0 / 1000: loss 1018.511824
iteration 100 / 1000: loss 7.433010
iteration 200 / 1000: loss 5.621980
iteration 300 / 1000: loss 5.592084
iteration 400 / 1000: loss 5.755761
iteration 500 / 1000: loss 6.178772
iteration 600 / 1000: loss 5.798188
iteration 700 / 1000: loss 6.436308
iteration 800 / 1000: loss 5.573876
iteration 900 / 1000: loss 5.936342
iteration 0 / 1000: loss 1097.606115
iteration 100 / 1000: loss 7.023526
iteration 200 / 1000: loss 5.599602
iteration 300 / 1000: loss 6.109078
iteration 400 / 1000: loss 5.501371
iteration 500 / 1000: loss 5.731113
iteration 600 / 1000: loss 5.815330
iteration 700 / 1000: loss 6.198909
iteration 800 / 1000: loss 6.145668
iteration 900 / 1000: loss 6.469148
iteration 0 / 1000: loss 1164.839431
iteration 100 / 1000: loss 7.285255
iteration 200 / 1000: loss 5.887818
iteration 300 / 1000: loss 6.319478
iteration 400 / 1000: loss 6.000932
iteration 500 / 1000: loss 5.760083
iteration 600 / 1000: loss 5.876752
iteration 700 / 1000: loss 5.652917
iteration 800 / 1000: loss 5.923112
iteration 900 / 1000: loss 4.895367
iteration 0 / 1000: loss 1247.806635
iteration 100 / 1000: loss 7.002475
iteration 200 / 1000: loss 6.332968
iteration 300 / 1000: loss 6.391068
iteration 400 / 1000: loss 5.830417
iteration 500 / 1000: loss 6.242772
iteration 600 / 1000: loss 6.185423
iteration 700 / 1000: loss 6.245708
iteration 800 / 1000: loss 5.836198
iteration 900 / 1000: loss 6.221194
iteration 0 / 1000: loss 1318.511392
iteration 100 / 1000: loss 6.529041
iteration 200 / 1000: loss 5.975797
iteration 300 / 1000: loss 5.860823
iteration 400 / 1000: loss 5.610748
iteration 500 / 1000: loss 5.780388
iteration 600 / 1000: loss 5.832203
iteration 700 / 1000: loss 6.642333
iteration 800 / 1000: loss 5.913785
iteration 900 / 1000: loss 6.105428
iteration 0 / 1000: loss 1409.076911
iteration 100 / 1000: loss 6.364561
iteration 200 / 1000: loss 5.645220
iteration 300 / 1000: loss 6.087972
iteration 400 / 1000: loss 6.471071
iteration 500 / 1000: loss 6.223955
iteration 600 / 1000: loss 6.280700
iteration 700 / 1000: loss 5.782700
iteration 800 / 1000: loss 5.670588
iteration 900 / 1000: loss 6.512198
```

```
iteration 0 / 1000: loss 1492.053502
iteration 100 / 1000: loss 6.001123
iteration 200 / 1000: loss 5.951767
iteration 300 / 1000: loss 5.593750
iteration 400 / 1000: loss 5.654671
iteration 500 / 1000: loss 6.196481
iteration 600 / 1000: loss 5.822579
iteration 700 / 1000: loss 5.550737
iteration 800 / 1000: loss 5.829225
iteration 900 / 1000: loss 5.496446
iteration 0 / 1000: loss 781.941194
iteration 100 / 1000: loss 6.899606
iteration 200 / 1000: loss 7.556690
iteration 300 / 1000: loss 6.692431
iteration 400 / 1000: loss 6.931333
iteration 500 / 1000: loss 6.275832
iteration 600 / 1000: loss 6.802207
iteration 700 / 1000: loss 5.672199
iteration 800 / 1000: loss 7.288850
iteration 900 / 1000: loss 6.422704
iteration 0 / 1000: loss 867.639133
iteration 100 / 1000: loss 7.752556
iteration 200 / 1000: loss 6.095638
iteration 300 / 1000: loss 6.598669
iteration 400 / 1000: loss 6.063491
iteration 500 / 1000: loss 7.489558
iteration 600 / 1000: loss 6.612219
iteration 700 / 1000: loss 6.835182
iteration 800 / 1000: loss 6.729848
iteration 900 / 1000: loss 5.690417
iteration 0 / 1000: loss 947.617568
iteration 100 / 1000: loss 6.787480
iteration 200 / 1000: loss 8.104288
iteration 300 / 1000: loss 6.423975
iteration 400 / 1000: loss 6.168763
iteration 500 / 1000: loss 7.006373
iteration 600 / 1000: loss 6.397417
iteration 700 / 1000: loss 5.937569
iteration 800 / 1000: loss 6.134733
iteration 900 / 1000: loss 6.591611
iteration 0 / 1000: loss 1016.772621
iteration 100 / 1000: loss 6.415656
iteration 200 / 1000: loss 7.290102
iteration 300 / 1000: loss 7.706845
iteration 400 / 1000: loss 7.407368
iteration 500 / 1000: loss 8.631588
iteration 600 / 1000: loss 7.169343
iteration 700 / 1000: loss 6.429326
iteration 800 / 1000: loss 7.102279
iteration 900 / 1000: loss 8.462611
iteration 0 / 1000: loss 1093.350494
iteration 100 / 1000: loss 7.246148
iteration 200 / 1000: loss 7.153807
iteration 300 / 1000: loss 6.924112
iteration 400 / 1000: loss 6.114679
iteration 500 / 1000: loss 6.335470
iteration 600 / 1000: loss 6.711672
iteration 700 / 1000: loss 6.358433
iteration 800 / 1000: loss 7.545562
iteration 900 / 1000: loss 7.029713
iteration 0 / 1000: loss 1170.763834
iteration 100 / 1000: loss 6.425543
iteration 200 / 1000: loss 6.194260
iteration 300 / 1000: loss 6.779903
iteration 400 / 1000: loss 6.786395
iteration 500 / 1000: loss 7.312726
iteration 600 / 1000: loss 5.856377
iteration 700 / 1000: loss 7.754485
iteration 800 / 1000: loss 6.939267
iteration 900 / 1000: loss 7.417401
iteration 0 / 1000: loss 1242.387005
iteration 100 / 1000: loss 7.697718
iteration 200 / 1000: loss 6.777957
iteration 300 / 1000: loss 6.515875
iteration 400 / 1000: loss 6.445109
iteration 500 / 1000: loss 7.583614
iteration 600 / 1000: loss 6.057816
```

```
iteration 700 / 1000: loss 7.716207
iteration 800 / 1000: loss 6.746935
iteration 900 / 1000: loss 7.728557
iteration 0 / 1000: loss 1317.267682
iteration 100 / 1000: loss 6.812326
iteration 200 / 1000: loss 7.729458
iteration 300 / 1000: loss 8.018226
iteration 400 / 1000: loss 6.280097
iteration 500 / 1000: loss 6.396834
iteration 600 / 1000: loss 7.170555
iteration 700 / 1000: loss 7.171397
iteration 800 / 1000: loss 7.142730
iteration 900 / 1000: loss 6.381536
iteration 0 / 1000: loss 1393.643099
iteration 100 / 1000: loss 6.273477
iteration 200 / 1000: loss 7.098653
iteration 300 / 1000: loss 5.915737
iteration 400 / 1000: loss 7.439607
iteration 500 / 1000: loss 7.646571
iteration 600 / 1000: loss 7.002962
iteration 700 / 1000: loss 7.334968
iteration 800 / 1000: loss 8.168845
iteration 900 / 1000: loss 6.231910
iteration 0 / 1000: loss 1486.490646
iteration 100 / 1000: loss 6.569873
iteration 200 / 1000: loss 6.746982
iteration 300 / 1000: loss 6.043391
iteration 400 / 1000: loss 6.447272
iteration 500 / 1000: loss 6.489602
iteration 600 / 1000: loss 7.683568
iteration 700 / 1000: loss 8.113927
iteration 800 / 1000: loss 7.397001
iteration 900 / 1000: loss 6.803898
iteration 0 / 1000: loss 784.453310
iteration 100 / 1000: loss 9.965764
iteration 200 / 1000: loss 12.356434
iteration 300 / 1000: loss 10.375885
iteration 400 / 1000: loss 13.292147
iteration 500 / 1000: loss 11.625326
iteration 600 / 1000: loss 11.702381
iteration 700 / 1000: loss 11.499117
iteration 800 / 1000: loss 11.348656
iteration 900 / 1000: loss 10.326493
iteration 0 / 1000: loss 853.819734
iteration 100 / 1000: loss 11.266343
iteration 200 / 1000: loss 10.971835
iteration 300 / 1000: loss 9.484434
iteration 400 / 1000: loss 12.908033
iteration 500 / 1000: loss 8.852770
iteration 600 / 1000: loss 8.211988
iteration 700 / 1000: loss 10.220348
iteration 800 / 1000: loss 12.460055
iteration 900 / 1000: loss 9.894184
iteration 0 / 1000: loss 943.288758
iteration 100 / 1000: loss 12.596178
iteration 200 / 1000: loss 10.680886
iteration 300 / 1000: loss 7.703995
iteration 400 / 1000: loss 13.020770
iteration 500 / 1000: loss 9.777482
iteration 600 / 1000: loss 11.522623
iteration 700 / 1000: loss 9.431546
iteration 800 / 1000: loss 8.609555
iteration 900 / 1000: loss 11.813485
iteration 0 / 1000: loss 1018.373372
iteration 100 / 1000: loss 11.718269
iteration 200 / 1000: loss 10.396942
iteration 300 / 1000: loss 8.179156
iteration 400 / 1000: loss 11.070975
iteration 500 / 1000: loss 12.483813
iteration 600 / 1000: loss 11.024602
iteration 700 / 1000: loss 9.497530
iteration 800 / 1000: loss 12.678874
iteration 900 / 1000: loss 12.241775
iteration 0 / 1000: loss 1096.994242
iteration 100 / 1000: loss 11.594937
iteration 200 / 1000: loss 13.638422
iteration 300 / 1000: loss 11.584809
```

```
iteration 400 / 1000: loss 13.215743
iteration 500 / 1000: loss 11.035672
iteration 600 / 1000: loss 8.032523
iteration 700 / 1000: loss 12.574125
iteration 800 / 1000: loss 11.478264
iteration 900 / 1000: loss 11.256982
iteration 0 / 1000: loss 1189.252586
iteration 100 / 1000: loss 12.825051
iteration 200 / 1000: loss 7.918515
iteration 300 / 1000: loss 12.061988
iteration 400 / 1000: loss 10.864221
iteration 500 / 1000: loss 16.651588
iteration 600 / 1000: loss 12.330111
iteration 700 / 1000: loss 12.100935
iteration 800 / 1000: loss 9.529177
iteration 900 / 1000: loss 14.461525
iteration 0 / 1000: loss 1245.685401
iteration 100 / 1000: loss 12.270377
iteration 200 / 1000: loss 11.550868
iteration 300 / 1000: loss 14.933232
iteration 400 / 1000: loss 14.967520
iteration 500 / 1000: loss 10.063976
iteration 600 / 1000: loss 13.105214
iteration 700 / 1000: loss 11.017074
iteration 800 / 1000: loss 9.020658
iteration 900 / 1000: loss 11.438681
iteration 0 / 1000: loss 1317.231357
iteration 100 / 1000: loss 12.939956
iteration 200 / 1000: loss 11.607363
iteration 300 / 1000: loss 11.807573
iteration 400 / 1000: loss 11.888326
iteration 500 / 1000: loss 12.981204
iteration 600 / 1000: loss 10.406423
iteration 700 / 1000: loss 10.853990
iteration 800 / 1000: loss 10.888479
iteration 900 / 1000: loss 9.583281
iteration 0 / 1000: loss 1386.771088
iteration 100 / 1000: loss 14.327519
iteration 200 / 1000: loss 12.595278
iteration 300 / 1000: loss 8.658656
iteration 400 / 1000: loss 11.105361
iteration 500 / 1000: loss 10.224135
iteration 600 / 1000: loss 11.098777
iteration 700 / 1000: loss 12.883108
iteration 800 / 1000: loss 13.761033
iteration 900 / 1000: loss 11.426378
iteration 0 / 1000: loss 1484.002325
iteration 100 / 1000: loss 14.189547
iteration 200 / 1000: loss 13.248536
iteration 300 / 1000: loss 11.074076
iteration 400 / 1000: loss 10.188821
iteration 500 / 1000: loss 10.559989
iteration 600 / 1000: loss 13.795688
iteration 700 / 1000: loss 12.744083
iteration 800 / 1000: loss 10.760855
iteration 900 / 1000: loss 12.093102
iteration 0 / 1000: loss 785.842881
iteration 100 / 1000: loss 16.278209
iteration 200 / 1000: loss 31.129586
iteration 300 / 1000: loss 21.015882
iteration 400 / 1000: loss 25.388158
iteration 500 / 1000: loss 17.981305
iteration 600 / 1000: loss 22.256400
iteration 700 / 1000: loss 32.273981
iteration 800 / 1000: loss 14.692289
iteration 900 / 1000: loss 23.048090
iteration 0 / 1000: loss 870.077079
iteration 100 / 1000: loss 20.783155
iteration 200 / 1000: loss 29.722553
iteration 300 / 1000: loss 23.305515
iteration 400 / 1000: loss 19.111394
iteration 500 / 1000: loss 21.159982
iteration 600 / 1000: loss 23.407995
iteration 700 / 1000: loss 26.779436
iteration 800 / 1000: loss 21.636624
iteration 900 / 1000: loss 25.128033
iteration 0 / 1000: loss 947.516479
```

```
iteration 100 / 1000: loss 21.942406
iteration 200 / 1000: loss 25.418684
iteration 300 / 1000: loss 18.343634
iteration 400 / 1000: loss 14.768238
iteration 500 / 1000: loss 19.408084
iteration 600 / 1000: loss 21.961284
iteration 700 / 1000: loss 23.289231
iteration 800 / 1000: loss 23.072451
iteration 900 / 1000: loss 29.949304
iteration 0 / 1000: loss 1027.219836
iteration 100 / 1000: loss 17.089522
iteration 200 / 1000: loss 22.643092
iteration 300 / 1000: loss 25.137248
iteration 400 / 1000: loss 26.859392
iteration 500 / 1000: loss 20.686417
iteration 600 / 1000: loss 19.686292
iteration 700 / 1000: loss 20.740670
iteration 800 / 1000: loss 20.696475
iteration 900 / 1000: loss 24.904107
iteration 0 / 1000: loss 1096.037853
iteration 100 / 1000: loss 20.485116
iteration 200 / 1000: loss 30.833928
iteration 300 / 1000: loss 22.241052
iteration 400 / 1000: loss 26.200192
iteration 500 / 1000: loss 20.875160
iteration 600 / 1000: loss 18.648409
iteration 700 / 1000: loss 33.743347
iteration 800 / 1000: loss 21.575968
iteration 900 / 1000: loss 18.589827
iteration 0 / 1000: loss 1182.377931
iteration 100 / 1000: loss 23.425792
iteration 200 / 1000: loss 27.065083
iteration 300 / 1000: loss 25.173871
iteration 400 / 1000: loss 22.159694
iteration 500 / 1000: loss 25.428774
iteration 600 / 1000: loss 19.759911
iteration 700 / 1000: loss 29.586277
iteration 800 / 1000: loss 26.313627
iteration 900 / 1000: loss 24.338969
iteration 0 / 1000: loss 1257.400946
iteration 100 / 1000: loss 19.824371
iteration 200 / 1000: loss 29.714360
iteration 300 / 1000: loss 23.067732
iteration 400 / 1000: loss 20.419255
iteration 500 / 1000: loss 28.216940
iteration 600 / 1000: loss 22.837337
iteration 700 / 1000: loss 29.451621
iteration 800 / 1000: loss 27.386279
iteration 900 / 1000: loss 33.840469
iteration 0 / 1000: loss 1336.064788
iteration 100 / 1000: loss 24.483160
iteration 200 / 1000: loss 26.233041
iteration 300 / 1000: loss 20.554243
iteration 400 / 1000: loss 23.738042
iteration 500 / 1000: loss 31.163149
iteration 600 / 1000: loss 24.989571
iteration 700 / 1000: loss 28.042039
iteration 800 / 1000: loss 22.680107
iteration 900 / 1000: loss 25.556713
iteration 0 / 1000: loss 1386.382761
iteration 100 / 1000: loss 19.706865
iteration 200 / 1000: loss 30.346519
iteration 300 / 1000: loss 25.793192
iteration 400 / 1000: loss 21.913649
iteration 500 / 1000: loss 23.733751
iteration 600 / 1000: loss 28.610142
iteration 700 / 1000: loss 27.220576
iteration 800 / 1000: loss 20.437934
iteration 900 / 1000: loss 21.969926
iteration 0 / 1000: loss 1462.934221
iteration 100 / 1000: loss 27.155078
iteration 200 / 1000: loss 31.953403
iteration 300 / 1000: loss 26.059223
iteration 400 / 1000: loss 40.883315
iteration 500 / 1000: loss 30.131299
iteration 600 / 1000: loss 25.156764
iteration 700 / 1000: loss 25.319102
```

```
iteration 800 / 1000: loss 32.478171
iteration 900 / 1000: loss 24.248506
iteration 0 / 1000: loss 795.334348
iteration 100 / 1000: loss 57.100328
iteration 200 / 1000: loss 63.883591
iteration 300 / 1000: loss 48.140827
iteration 400 / 1000: loss 55.851625
iteration 500 / 1000: loss 48.829302
iteration 600 / 1000: loss 50.300895
iteration 700 / 1000: loss 68.712529
iteration 800 / 1000: loss 63.909238
iteration 900 / 1000: loss 62.279553
iteration 0 / 1000: loss 880.561380
iteration 100 / 1000: loss 45.944172
iteration 200 / 1000: loss 68.689149
iteration 300 / 1000: loss 61.636590
iteration 400 / 1000: loss 58.771024
iteration 500 / 1000: loss 46.791250
iteration 600 / 1000: loss 83.700232
iteration 700 / 1000: loss 60.457466
iteration 800 / 1000: loss 74.160595
iteration 900 / 1000: loss 56.046042
iteration 0 / 1000: loss 949.746613
iteration 100 / 1000: loss 50.247242
iteration 200 / 1000: loss 43.218012
iteration 300 / 1000: loss 73.874360
iteration 400 / 1000: loss 63.116916
iteration 500 / 1000: loss 94.770185
iteration 600 / 1000: loss 57.666982
iteration 700 / 1000: loss 83.958730
iteration 800 / 1000: loss 82.209782
iteration 900 / 1000: loss 50.658497
iteration 0 / 1000: loss 1016.047606
iteration 100 / 1000: loss 101.443292
iteration 200 / 1000: loss 52.463837
iteration 300 / 1000: loss 63.320347
iteration 400 / 1000: loss 85.665076
iteration 500 / 1000: loss 65.127090
iteration 600 / 1000: loss 74.192636
iteration 700 / 1000: loss 97.516331
iteration 800 / 1000: loss 53.463083
iteration 900 / 1000: loss 70.181272
iteration 0 / 1000: loss 1101.123822
iteration 100 / 1000: loss 88.630741
iteration 200 / 1000: loss 67.393761
iteration 300 / 1000: loss 56.074835
iteration 400 / 1000: loss 74.986379
iteration 500 / 1000: loss 90.327314
iteration 600 / 1000: loss 79.473833
iteration 700 / 1000: loss 67.305880
iteration 800 / 1000: loss 81.675328
iteration 900 / 1000: loss 95.058048
iteration 0 / 1000: loss 1176.654103
iteration 100 / 1000: loss 94.140512
iteration 200 / 1000: loss 88.180551
iteration 300 / 1000: loss 95.842167
iteration 400 / 1000: loss 83.366559
iteration 500 / 1000: loss 76.417351
iteration 600 / 1000: loss 96.312632
iteration 700 / 1000: loss 73.442088
iteration 800 / 1000: loss 83.989990
iteration 900 / 1000: loss 83.757383
iteration 0 / 1000: loss 1248.885020
iteration 100 / 1000: loss 67.263235
iteration 200 / 1000: loss 81.853817
iteration 300 / 1000: loss 94.123600
iteration 400 / 1000: loss 143.951995
iteration 500 / 1000: loss 98.230871
iteration 600 / 1000: loss 77.278612
iteration 700 / 1000: loss 98.261396
iteration 800 / 1000: loss 117.254297
iteration 900 / 1000: loss 98.308607
iteration 0 / 1000: loss 1334.195122
iteration 100 / 1000: loss 98.341535
iteration 200 / 1000: loss 97.371864
iteration 300 / 1000: loss 78.699611
iteration 400 / 1000: loss 86.144391
```

```
iteration 500 / 1000: loss 106.652220
iteration 600 / 1000: loss 59.992439
iteration 700 / 1000: loss 75.884815
iteration 800 / 1000: loss 107.554304
iteration 900 / 1000: loss 83.475420
iteration 0 / 1000: loss 1414.161606
iteration 100 / 1000: loss 82.978844
iteration 200 / 1000: loss 115.607573
iteration 300 / 1000: loss 103.305578
iteration 400 / 1000: loss 95.082451
iteration 500 / 1000: loss 92.799051
iteration 600 / 1000: loss 119.995194
iteration 700 / 1000: loss 110.400711
iteration 800 / 1000: loss 93.144908
iteration 900 / 1000: loss 123.195199
iteration 0 / 1000: loss 1476.985959
iteration 100 / 1000: loss 137.994563
iteration 200 / 1000: loss 100.068277
iteration 300 / 1000: loss 115.836155
iteration 400 / 1000: loss 97.025609
iteration 500 / 1000: loss 124.645036
iteration 600 / 1000: loss 108.345365
iteration 700 / 1000: loss 109.061558
iteration 800 / 1000: loss 95.736110
iteration 900 / 1000: loss 89.458356
iteration 0 / 1000: loss 794.441967
iteration 100 / 1000: loss 388.715293
iteration 200 / 1000: loss 374.607415
iteration 300 / 1000: loss 331.678067
iteration 400 / 1000: loss 317.892843
iteration 500 / 1000: loss 295.940212
iteration 600 / 1000: loss 307.128730
iteration 700 / 1000: loss 266.641776
iteration 800 / 1000: loss 301.700302
iteration 900 / 1000: loss 299.543973
iteration 0 / 1000: loss 870.601522
iteration 100 / 1000: loss 339.512232
iteration 200 / 1000: loss 281.351848
iteration 300 / 1000: loss 354.650584
iteration 400 / 1000: loss 431.562670
iteration 500 / 1000: loss 355.064072
iteration 600 / 1000: loss 440.575192
iteration 700 / 1000: loss 416.136467
iteration 800 / 1000: loss 381.243188
iteration 900 / 1000: loss 451.356119
iteration 0 / 1000: loss 948.250353
iteration 100 / 1000: loss 473.784222
iteration 200 / 1000: loss 401.916628
iteration 300 / 1000: loss 517.014151
iteration 400 / 1000: loss 437.976712
iteration 500 / 1000: loss 511.612816
iteration 600 / 1000: loss 538.431093
iteration 700 / 1000: loss 609.749092
iteration 800 / 1000: loss 592.342878
iteration 900 / 1000: loss 589.016043
iteration 0 / 1000: loss 1021.366239
iteration 100 / 1000: loss 591.975484
iteration 200 / 1000: loss 606.669151
iteration 300 / 1000: loss 691.130107
iteration 400 / 1000: loss 624.652513
iteration 500 / 1000: loss 592.264926
iteration 600 / 1000: loss 586.382451
iteration 700 / 1000: loss 601.377355
iteration 800 / 1000: loss 680.185347
iteration 900 / 1000: loss 607.962779
iteration 0 / 1000: loss 1093.567366
iteration 100 / 1000: loss 852.910709
iteration 200 / 1000: loss 915.618730
iteration 300 / 1000: loss 966.376486
iteration 400 / 1000: loss 1093.374193
iteration 500 / 1000: loss 919.035570
iteration 600 / 1000: loss 907.350024
iteration 700 / 1000: loss 905.366154
iteration 800 / 1000: loss 1032.082805
iteration 900 / 1000: loss 1087.302172
iteration 0 / 1000: loss 1178.453096
```

```
iteration 100 / 1000: loss 1252.401445
iteration 200 / 1000: loss 1882.848777
iteration 300 / 1000: loss 1430.729259
iteration 400 / 1000: loss 1612.657978
iteration 500 / 1000: loss 1441.985971
iteration 600 / 1000: loss 1475.858914
iteration 700 / 1000: loss 1231.579424
iteration 800 / 1000: loss 1425.628694
iteration 900 / 1000: loss 1546.048011
iteration 0 / 1000: loss 1240.311256
iteration 100 / 1000: loss 2560.224755
iteration 200 / 1000: loss 2966.166408
iteration 300 / 1000: loss 2643.692542
iteration 400 / 1000: loss 2503.925170
iteration 500 / 1000: loss 2788.497377
iteration 600 / 1000: loss 2557.271196
iteration 700 / 1000: loss 2788.739355
iteration 800 / 1000: loss 2555.820183
iteration 900 / 1000: loss 2625.299711
iteration 0 / 1000: loss 1332.926114
iteration 100 / 1000: loss 7453.175896
iteration 200 / 1000: loss 7506.425897
iteration 300 / 1000: loss 6903.300192
iteration 400 / 1000: loss 7196.690224
iteration 500 / 1000: loss 6483.055431
iteration 600 / 1000: loss 6430.599134
iteration 700 / 1000: loss 7026.176787
iteration 800 / 1000: loss 6597.479706
iteration 900 / 1000: loss 6659.290954
iteration 0 / 1000: loss 1407.774238
iteration 100 / 1000: loss 50071.713043
iteration 200 / 1000: loss 48179.435256
iteration 300 / 1000: loss 47498.169873
iteration 400 / 1000: loss 48080.966030
iteration 500 / 1000: loss 47639.830622
iteration 600 / 1000: loss 46792.206403
iteration 700 / 1000: loss 47966.460749
iteration 800 / 1000: loss 45917.823284
iteration 900 / 1000: loss 46574.005228
iteration 0 / 1000: loss 1494.583867
iteration 100 / 1000: loss 724017626.980267
iteration 200 / 1000: loss 6823568611928.416992
iteration 300 / 1000: loss 63040831938746736.000000
iteration 400 / 1000: loss 582294666030022393856.000000
iteration 500 / 1000: loss 5378519463616464239460352.000000
iteration 600 / 1000: loss 496801235993401281036828016 64.000000
iteration 700 / 1000: loss 458883656920637128295179175854080.000000
iteration 800 / 1000: loss 423860077898751780753130295907 2673792.000000
iteration 900 / 1000: loss 391509618891054842294694155898161631 06816.000000
iteration 0 / 1000: loss 789.875106
iteration 100 / 1000: loss 667793893457653628087880 9088.000000
iteration 200 / 1000: loss 98104138798836159600159691208873499909517898625843 20.000000
iteration 300 / 1000: loss
144122642386225140179326067464513169978816160089492110349803769350139339079 68.000000
iteration 400 / 1000: loss
211727418462738221421097232125029880270525913619795468437230309076852380383484767979339916761513328
00000
iteration 500 / 1000: loss
311044114836323906366458012488575614454931665266617490692406818591686086750637079345648023761949235
2285520866439757561856.000000
iteration 600 / 1000: loss
456948099007493189392711678559559975860387321662035906511004306630885406923693058291490487695170727
241629689906733762324767049038747295610725990 4.000000
iteration 700 / 1000: loss
671292447685230604578169850932339442984150934736472243772509051404068813904669325351025215534233636
192649881889621860243983058281828907326149522305483309283086597881856 0.000000
iteration 800 / 1000: loss
986181037404509396014285911238238814092807839160366383348612711929491279362413652001387893055906069
982146253056365173030273569787272471870949055726697569725077483976064174186038871237148121497 6.0000

iteration 900 / 1000: loss
144877697029039864733594926276523940028478637267396717672587491994049199033047462801611318164153259
837694047929443310323931187631906100925846455140417635200981734519852855600291068475413328317079265
2334049522089984.000000
iteration 0 / 1000: loss 865.157168
iteration 100 / 1000: loss 335420140856814554300645834948326444761088.000000
iteration 200 / 1000: loss
566011080608461604935631419328885424670802206941128513722704478785807123082117 12.000000
```

```
iteration 300 / 1000: loss
955126136889670373184054007229474313344060994202963738327880687688076592119655364907133534681880221
678655954976768.000000
iteration 400 / 1000: loss
161174572128358329825817718476617802504641878065693003329669825735851655313409868339260429315408541
171747093201799954327370203954909899965173534319181824.000000
iteration 500 / 1000: loss
271977089699934568241556612237169768757459742908679917179651293547462952211240642961533118633054708
923592652613222826896293120986814188908913974430002215462120771812284705874410995656299642880.000000

iteration 600 / 1000: loss
458952900230042694182974437863570006271292430513812228555695299994764311795566580118585906992441116
467957219419771626406899350816213211545556306585260638905399121092248015495461180176728376692112722
148411952992256578864283648.000000
iteration 700 / 1000: loss
774468779197390472619347537781343697109705192628840349095758375670517268830270565706309420656440653
909741960412267378254667669161006471679693730785707456722169816967910731746610226425768527565146415
401909176277065759472605558783288951344472737316851442267245772800.000000
```

```
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:102:
RuntimeWarning: overflow encountered in double_scalars
  loss = np.sum(Loss_i, axis=0) / N_train + reg*np.sum(W*W)
C:\Users\Jnani\Anaconda3\lib\site-packages\numpy\core\fromnumeric.py:83: RuntimeWarning: overflow
encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:102:
RuntimeWarning: overflow encountered in multiply
  loss = np.sum(Loss_i, axis=0) / N_train + reg*np.sum(W*W)
```

```
iteration 800 / 1000: loss inf
iteration 900 / 1000: loss inf
iteration 0 / 1000: loss 945.408052
iteration 100 / 1000: loss 37384509788417153693209687188536790155976694793502720 0.000000
iteration 200 / 1000: loss
790793908887674074422565685452740258528039274204316225268247333374558770809807604861861214375530777
0.000000
iteration 300 / 1000: loss
167276502988304719671485384124606832706022533693061426565046088537964898852631622818050913613504013
212146990934274393577123358764216321232078075068416 0.000000
iteration 400 / 1000: loss
353839706369954550417870087964073300140997262284642126444718738911650462418489580506133346626739497
900869767419169026756811763553853775157822322233275076842329500346754174580434697443813383784720498
000000
iteration 500 / 1000: loss
748476537752162951742183639740470263429756106689985459585830478731658107733720244289084827074482957
905183568438400443393828038349842438232413578294503494062634421065434975613351898292926216550639686
480903745600428697593881033643088686928049366630 4.000000
iteration 600 / 1000: loss
158325116565559774973370651813446828379479544346009912543415889180259969320594010668316530818796571
024885055078181946139461560626967481507331541229651371295474624656217842824944917071923282587590224
478819600510672246743990045225681536256244968244864132319703259791857617138602777314594416275698483
0000
iteration 700 / 1000: loss inf
iteration 800 / 1000: loss inf
iteration 900 / 1000: loss inf
iteration 0 / 1000: loss 1015.124101
iteration 100 / 1000: loss
120169059847648734484467668765058814305223882197313796821913436 16.000000
iteration 200 / 1000: loss
105288822838806497596130831858423395193524560786121166868446288223487749139687858377993300868191146
009075890315882177495 04.000000
iteration 300 / 1000: loss
922511687187714393758879252357946289095410652331684267800696461696426396980982715408366069183535023
079471574004571151621920861804600365445725365925632162388406805896834444214780559 36.000000
iteration 400 / 1000: loss
808279350127047263933027336915594382871240934746590808588447681555646902997162524973899417407444106
614158356080725681561282472416589471483155941230243157669174784112124904489436223024106569946940982
838736957295880541396113650636840039874 56.000000
iteration 500 / 1000: loss
708192120398431272173696950296799611603318982903975692540325056658835467081822493567443055656979660
548599769299119395378945160124271751594429000486043169518939310259377155928937867144834486104077122
913471499074803821132943707649220197618011917189633459867908027771653488856683689564585684859218821
000000
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss inf
iteration 800 / 1000: loss inf
```

```
iteration 900 / 1000: loss inf
iteration 0 / 1000: loss 1099.088622
iteration 100 / 1000: loss
380680167048297155690421729917092232573968804448933434634799029038182563 84.000000
iteration 200 / 1000: loss
961133873976563843385880208300154082215577686909451772139303265941871843355745526570302486201131744
301141258221481358240824838234543069593 60.000000
iteration 300 / 1000: loss
242665209188057520080865922077836762638737636983761672409074989786685296621040782280702833781711457
568703713660781139006125739524810152859892864294069209166291242220943370276907895213484300043357963
735476224.000000
iteration 400 / 1000: loss
612676395502002797930949629697668982549600882217097045339814864741959729540910222500686589580530886
409979468356267573972144934223551483909155950999652744441190580097624251460230195632880928161912371
708159777850813020739460421656672638410872226864524864463178810328889540424499 2.000000
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss inf
iteration 800 / 1000: loss inf
```

```
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:93:
RuntimeWarning: overflow encountered in subtract
  score_diffs = scores - np.expand_dims(correct_scores, -1) + 1 #add delta=1
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:132:
RuntimeWarning: overflow encountered in multiply
  dW += 2*reg*W
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:93:
RuntimeWarning: invalid value encountered in subtract
  score_diffs = scores - np.expand_dims(correct_scores, -1) + 1 #add delta=1
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:96:
RuntimeWarning: invalid value encountered in maximum
  score_diffs_capped = np.maximum(np.zeros(score_diffs.shape),score_diffs)
C:\Users\Jnani\Desktop\classwork\hw1\assignment1\cs231n\classifiers\linear_svm.py:121:
RuntimeWarning: invalid value encountered in greater
  dw_col[score_diffs_capped>0] = 1 #representative of the result of the "if margin>0" above
```

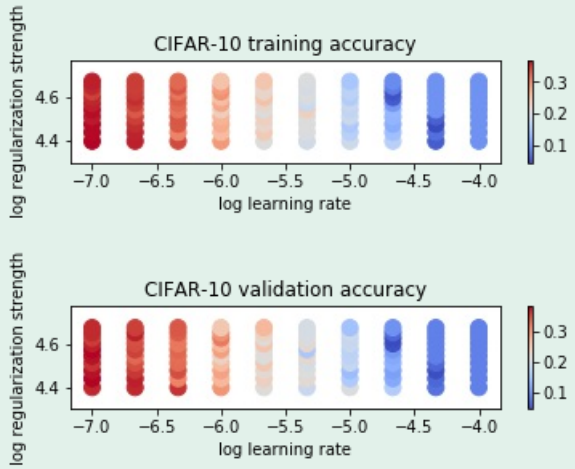
```
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1176.078912
iteration 100 / 1000: loss
119621247360543116854385882956578003249400084121873042393400593654813668074302996 48.000000
iteration 200 / 1000: loss
102298590552064777562128762238039219299842852231925236402566362330358004240436160241916011340827208
467844333484062012229756423324612575556360166836075516395 52.000000
iteration 300 / 1000: loss
874844716958775047692968267899853927802147970224496559049543341757553908631003104748252229785738361
197961333734101838903541853754050578578722267864373124775849134088900753601523970187856855964267959
161938710471861680270547302010060 8.000000
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss inf
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1250.953384
iteration 100 / 1000: loss
744145002760271201717377830109461502443553882190016970966554877691594074914667884160483328.000000
iteration 200 / 1000: loss
371787098299079150303510499817749120757424528651340994998272142721511233896771874271591562147851341
115679094492193970214009337953144928379917825031020646729025033395670876 16.000000
iteration 300 / 1000: loss
185750957070095356433233424637730481605845211026311813985733370786127326016816690962280616746926995
161642238926424054811694246109146166223069497182532974081541407412492825279520794526941041430425904
904375224260951177453624582219809757819468595751425146880 0.000000
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss inf
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1335.221145
iteration 100 / 1000: loss
100555316849175866791494514748116081157842274980803846639985556211397964064009125697893897317908 48.
00
iteration 200 / 1000: loss
675774868909473742190413158612442014020176244929342971168226006727877593209256048997383087749494114
930067492713145709727311652755458040561513284524395598913866887126886783596407812836 5568.000000
iteration 300 / 1000: loss
```

```
iteration 300 / 1000: loss
45414970362490523944190204150565794966464159052040932315130886240316991937091523117154652380146022434860603125809225900800949696763103337892408510321416091366317875469879547155314346581572092062579376996993264355510087793877930104898504554328489116980337335610779617811035586560.000000
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1395.217878
iteration 100 / 1000: loss
39425596657378065970669930997610038527886265556862150598139170022348114646149559347363045437755220888.000000
iteration 200 / 1000: loss
10255934679682831146488628204842462031366799969807681337156719074453221804491654514410935608660236087683631482050148630615925428380218618510989968015094046825440636493499244896396994315672639787080000000
iteration 300 / 1000: loss
26679164063896775642819703207867456692327419274891853389117558883615737029488571211754342576150556812070043802824504658223453485547595873718751761892884305726638996156903174074312706717141199255873469804962846511002384217815444169431267281270400441421421970667483751362599714646941220760801666662400000
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss inf
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1488.064182
iteration 100 / 1000: loss
55360502975502784832689130483050469269737282765960822586787693749449451600254488454417212833181356711398784.000000
iteration 200 / 1000: loss
19123978540539969562321541464567341908043468990080872417452668240419632940847375727133406530161799399319412529693288840859617724035528466113205410757203684341529976470475694491992355599080213211908619455922176.000000
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 790.618543
iteration 100 / 1000: loss
23970318231057778635588022311589573089661389137171918308857715433928841327074733967406054803676528074713100249497128140 8.000000
iteration 200 / 1000: loss
61897351329853351243762975877820871734220160325552890349714973601313195888458797884131397296669340142657452942580743414284757303679087650221803962843129881653861041519732179128675431661634949089049995382912401956281618608027387119534080.000000
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss inf
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 871.641402
iteration 100 / 1000: loss
46326297257529791157896873226251659365842455076687918026212234551401163599732379646272960674439870614932713487189793493775024128000.000000
iteration 200 / 1000: loss
20339035044358114566266275490599402652471264009443646761696277207085558945793884753930896115909445577962268075249644512700552938649488488233295808412309953398892580766523132748791496572539549294089690004745276980225485998574112346879418005695990185448177664.000000
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 946.305878
iteration 100 / 1000: loss
68500959607456290616046961453744052296972949808602026779403777972564084481364139193352782046190628787062902148437995860173314539791916728320.000000
```

```
iteration 200 / 1000: loss
426282518185354477226222318357293838200988113280722676775304195470250573083776014460914213291073710
379681152743291152122918839547870469918609311295754515170069098076531158126312573043489926330593539
381529060093783347637810912177892689032170487551279866520848920138297617940 48.000000
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1001.421156
iteration 100 / 1000: loss
129952172344466026194110686196054091467089901785850331039629619678605411536358271057708877054722143
455184087884535740830785118822293241536219552153 6.000000
iteration 200 / 1000: loss
153575324847659358604409315030041384913126040701477311209930910411690971238858439685441759753127165
533341753774586138330262767214567497242988208469111393304804383274647533110793028672503306352333905
432069807156543011954702221305800142021215513334132871469451495627240141026895341386357066956 8.0000

iteration 300 / 1000: loss inf
iteration 400 / 1000: loss inf
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1097.678743
iteration 100 / 1000: loss
516352119085917249498174939669847389304919088442882538988704558181711827394298324815226940490827109
071685718222941936474007033938575622910221073997512048 64.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1175.401502
iteration 100 / 1000: loss
502787499503526486771152064930439255947993286773026899031530878247227538183428244870024616660190112
209438350123653641540446968112780595200279706496004086615244 800.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1264.758392
iteration 100 / 1000: loss
143273168639118812806862196624341391613749097337442753904961258830174439742088261263820361071623970
411280209387820104902431539623450367065690408482023891236183464319385 6.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1337.222378
iteration 100 / 1000: loss
145555701404680929977920131238137712260798105693192772218280657330493519727884073273198695447239612
471928030278807320875344296093031946429068829286288840490584382578761295462 4.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1399.069280
iteration 100 / 1000: loss
```

```
iteration 100 / 1000: loss
614528158789768516317186207871718766685255147593988251899631224544589929734706384738950533299461075
725159282390883460763610045722448657732184662786216695863524610261678002981044224.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
iteration 0 / 1000: loss 1470.502282
iteration 100 / 1000: loss
117033464574611252456423396432764094301162143698950346629304075850556696884012645748980116195518064
550917651951895372353461059923214292772344034870726115690250514136954586778078153801728.000000
iteration 200 / 1000: loss inf
iteration 300 / 1000: loss inf
iteration 400 / 1000: loss nan
iteration 500 / 1000: loss nan
iteration 600 / 1000: loss nan
iteration 700 / 1000: loss nan
iteration 800 / 1000: loss nan
iteration 900 / 1000: loss nan
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-ddc81826feed> in <module>
     59 # Print out results.
     60 for lr, rs in sorted(results):
---> 61     train_accuracy, val_accuracy = results[(lr, reg)]
     62     print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
     63                 lr, rs, train_accuracy, val_accuracy))

NameError: name 'reg' is not defined
```



In [17]:

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

```
linear SVM on raw pixels final test set accuracy: 0.354000
```

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:

The svm weights look like very blurry representatives of pictures containing that class of object, down even to the background color. The blurry and somewhat circular/centered look is likely an attempt by the model to build some position invariance into the templates to successfully classify the objects in images taken from various angles.

In [ ]:

```
```

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 [26]:

```
# 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: 2.331291
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer: Withg 10 classes, random chance gives a 1/10 or 0.1 chance to get the correct class, meaning that the "log probability" of the correct class, the softmax's loss function, should be about -log(0.1).

In [96]:

```
# 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: 1.659833 analytic: 1.659832, relative error: 3.367626e-08
numerical: -1.359423 analytic: -1.359423, relative error: 3.813230e-08
numerical: 0.262690 analytic: 0.262690, relative error: 7.807496e-08
numerical: 0.260485 analytic: 0.260485, relative error: 2.428047e-07
```

```
numerical: 3.186805 analytic: 3.186805, relative error: 1.529181e-08
numerical: -0.133671 analytic: -0.133671, relative error: 4.307653e-07
numerical: 1.465872 analytic: 1.465873, relative error: 1.407267e-08
numerical: 0.450397 analytic: 0.450397, relative error: 3.315398e-08
numerical: -1.619753 analytic: -1.619753, relative error: 1.698572e-08
numerical: -0.740496 analytic: -0.740496, relative error: 5.615728e-08
numerical: 0.210287 analytic: 0.210287, relative error: 1.618553e-08
numerical: 1.038310 analytic: 1.038310, relative error: 5.956410e-08
numerical: -2.159982 analytic: -2.159982, relative error: 9.671621e-09
numerical: -0.929786 analytic: -0.929786, relative error: 3.893478e-08
numerical: 2.992666 analytic: 2.992666, relative error: 1.987621e-08
numerical: 3.160100 analytic: 3.160100, relative error: 1.985018e-08
numerical: 2.033895 analytic: 2.033895, relative error: 4.995504e-09
numerical: -0.711310 analytic: -0.711310, relative error: 4.264022e-08
numerical: -4.717660 analytic: -4.717660, relative error: 2.321304e-08
numerical: 0.106126 analytic: 0.106125, relative error: 3.661198e-07
```

In [97]:

```
# 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: 2.331291e+00 computed in 0.156582s
vectorized loss: 2.331291e+00 computed in 0.002996s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [108]:

```
# 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, 5e-7]
regularization_strengths = [2.5e4, 5e4]

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in np.arange(learning_rates[0],learning_rates[1],np.diff(learning_rates)/10):
    for rs in np.arange(regularization_strengths[0],regularization_strengths[1],np.diff(regularizat
ion_strengths)/10):
        softm = Softmax()
        loss_hist = softm.train(X_train, y_train, learning_rate=lr, reg=rs,
                                num_iters=500, verbose=False)

        Yt = softm.predict(X_train)
        Yp = softm.predict(X_val)

        t_num_correct = np.sum(Yt == y_train)
```

```
        t_accuracy = float(t_num_correct) / X_train.shape[0]

        v_num_correct = np.sum(Yp == y_val)
        v_accuracy = float(v_num_correct) / X_val.shape[0]

        results[(lr,rs)] = (t_accuracy ,v_accuracy)

        if (best_val<v_accuracy):
            best_val = v_accuracy
            best_softmax = softm

# *****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 2.500000e+04 train accuracy: 0.313796 val accuracy: 0.327000
lr 1.000000e-07 reg 2.750000e+04 train accuracy: 0.315694 val accuracy: 0.318000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.315939 val accuracy: 0.327000
lr 1.000000e-07 reg 3.250000e+04 train accuracy: 0.315204 val accuracy: 0.333000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.310510 val accuracy: 0.324000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.314347 val accuracy: 0.336000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.313000 val accuracy: 0.320000
lr 1.000000e-07 reg 4.250000e+04 train accuracy: 0.314755 val accuracy: 0.323000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.305898 val accuracy: 0.322000
lr 1.000000e-07 reg 4.750000e+04 train accuracy: 0.305204 val accuracy: 0.326000
lr 1.400000e-07 reg 2.500000e+04 train accuracy: 0.322694 val accuracy: 0.334000
lr 1.400000e-07 reg 2.750000e+04 train accuracy: 0.317286 val accuracy: 0.334000
lr 1.400000e-07 reg 3.000000e+04 train accuracy: 0.320041 val accuracy: 0.337000
lr 1.400000e-07 reg 3.250000e+04 train accuracy: 0.316694 val accuracy: 0.331000
lr 1.400000e-07 reg 3.500000e+04 train accuracy: 0.324939 val accuracy: 0.333000
lr 1.400000e-07 reg 3.750000e+04 train accuracy: 0.309939 val accuracy: 0.322000
lr 1.400000e-07 reg 4.000000e+04 train accuracy: 0.312837 val accuracy: 0.325000
lr 1.400000e-07 reg 4.250000e+04 train accuracy: 0.311367 val accuracy: 0.319000
lr 1.400000e-07 reg 4.500000e+04 train accuracy: 0.310571 val accuracy: 0.321000
lr 1.400000e-07 reg 4.750000e+04 train accuracy: 0.306918 val accuracy: 0.325000
lr 1.800000e-07 reg 2.500000e+04 train accuracy: 0.336347 val accuracy: 0.352000
lr 1.800000e-07 reg 2.750000e+04 train accuracy: 0.331020 val accuracy: 0.347000
lr 1.800000e-07 reg 3.000000e+04 train accuracy: 0.331306 val accuracy: 0.347000
lr 1.800000e-07 reg 3.250000e+04 train accuracy: 0.318367 val accuracy: 0.335000
lr 1.800000e-07 reg 3.500000e+04 train accuracy: 0.321510 val accuracy: 0.338000
lr 1.800000e-07 reg 3.750000e+04 train accuracy: 0.314755 val accuracy: 0.331000
lr 1.800000e-07 reg 4.000000e+04 train accuracy: 0.320184 val accuracy: 0.336000
lr 1.800000e-07 reg 4.250000e+04 train accuracy: 0.306837 val accuracy: 0.321000
lr 1.800000e-07 reg 4.500000e+04 train accuracy: 0.313224 val accuracy: 0.318000
lr 1.800000e-07 reg 4.750000e+04 train accuracy: 0.310449 val accuracy: 0.326000
lr 2.200000e-07 reg 2.500000e+04 train accuracy: 0.328429 val accuracy: 0.341000
lr 2.200000e-07 reg 2.750000e+04 train accuracy: 0.322265 val accuracy: 0.339000
lr 2.200000e-07 reg 3.000000e+04 train accuracy: 0.316592 val accuracy: 0.340000
lr 2.200000e-07 reg 3.250000e+04 train accuracy: 0.324816 val accuracy: 0.341000
lr 2.200000e-07 reg 3.500000e+04 train accuracy: 0.314735 val accuracy: 0.335000
lr 2.200000e-07 reg 3.750000e+04 train accuracy: 0.311367 val accuracy: 0.338000
lr 2.200000e-07 reg 4.000000e+04 train accuracy: 0.306755 val accuracy: 0.329000
lr 2.200000e-07 reg 4.250000e+04 train accuracy: 0.311020 val accuracy: 0.318000
lr 2.200000e-07 reg 4.500000e+04 train accuracy: 0.302918 val accuracy: 0.321000
lr 2.200000e-07 reg 4.750000e+04 train accuracy: 0.305510 val accuracy: 0.312000
lr 2.600000e-07 reg 2.500000e+04 train accuracy: 0.326837 val accuracy: 0.343000
lr 2.600000e-07 reg 2.750000e+04 train accuracy: 0.327612 val accuracy: 0.336000
lr 2.600000e-07 reg 3.000000e+04 train accuracy: 0.319837 val accuracy: 0.329000
lr 2.600000e-07 reg 3.250000e+04 train accuracy: 0.316347 val accuracy: 0.332000
lr 2.600000e-07 reg 3.500000e+04 train accuracy: 0.316959 val accuracy: 0.333000
lr 2.600000e-07 reg 3.750000e+04 train accuracy: 0.308163 val accuracy: 0.327000
lr 2.600000e-07 reg 4.000000e+04 train accuracy: 0.301184 val accuracy: 0.311000
lr 2.600000e-07 reg 4.250000e+04 train accuracy: 0.314041 val accuracy: 0.333000
lr 2.600000e-07 reg 4.500000e+04 train accuracy: 0.306816 val accuracy: 0.321000
lr 2.600000e-07 reg 4.750000e+04 train accuracy: 0.308694 val accuracy: 0.310000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.324388 val accuracy: 0.340000
lr 3.000000e-07 reg 2.750000e+04 train accuracy: 0.317837 val accuracy: 0.330000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.326592 val accuracy: 0.335000
lr 3.000000e-07 reg 3.250000e+04 train accuracy: 0.322286 val accuracy: 0.338000
lr 3.000000e-07 reg 3.500000e+04 train accuracy: 0.321143 val accuracy: 0.326000
```

```
lr 3.000000e-07 reg 3.750000e+04 train accuracy: 0.316510 val accuracy: 0.328000
lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.316286 val accuracy: 0.326000
lr 3.000000e-07 reg 4.250000e+04 train accuracy: 0.316367 val accuracy: 0.334000
lr 3.000000e-07 reg 4.500000e+04 train accuracy: 0.304694 val accuracy: 0.318000
lr 3.000000e-07 reg 4.750000e+04 train accuracy: 0.313286 val accuracy: 0.321000
lr 3.400000e-07 reg 2.500000e+04 train accuracy: 0.326796 val accuracy: 0.335000
lr 3.400000e-07 reg 2.750000e+04 train accuracy: 0.327551 val accuracy: 0.335000
lr 3.400000e-07 reg 3.000000e+04 train accuracy: 0.315531 val accuracy: 0.329000
lr 3.400000e-07 reg 3.250000e+04 train accuracy: 0.316122 val accuracy: 0.336000
lr 3.400000e-07 reg 3.500000e+04 train accuracy: 0.317714 val accuracy: 0.334000
lr 3.400000e-07 reg 3.750000e+04 train accuracy: 0.317449 val accuracy: 0.327000
lr 3.400000e-07 reg 4.000000e+04 train accuracy: 0.310122 val accuracy: 0.328000
lr 3.400000e-07 reg 4.250000e+04 train accuracy: 0.317694 val accuracy: 0.323000
lr 3.400000e-07 reg 4.500000e+04 train accuracy: 0.296633 val accuracy: 0.316000
lr 3.400000e-07 reg 4.750000e+04 train accuracy: 0.307490 val accuracy: 0.327000
lr 3.800000e-07 reg 2.500000e+04 train accuracy: 0.324204 val accuracy: 0.332000
lr 3.800000e-07 reg 2.750000e+04 train accuracy: 0.320939 val accuracy: 0.344000
lr 3.800000e-07 reg 3.000000e+04 train accuracy: 0.324898 val accuracy: 0.332000
lr 3.800000e-07 reg 3.250000e+04 train accuracy: 0.311061 val accuracy: 0.332000
lr 3.800000e-07 reg 3.500000e+04 train accuracy: 0.320612 val accuracy: 0.343000
lr 3.800000e-07 reg 3.750000e+04 train accuracy: 0.314673 val accuracy: 0.324000
lr 3.800000e-07 reg 4.000000e+04 train accuracy: 0.299755 val accuracy: 0.317000
lr 3.800000e-07 reg 4.250000e+04 train accuracy: 0.314653 val accuracy: 0.328000
lr 3.800000e-07 reg 4.500000e+04 train accuracy: 0.315510 val accuracy: 0.332000
lr 3.800000e-07 reg 4.750000e+04 train accuracy: 0.295469 val accuracy: 0.317000
lr 4.200000e-07 reg 2.500000e+04 train accuracy: 0.318592 val accuracy: 0.355000
lr 4.200000e-07 reg 2.750000e+04 train accuracy: 0.320918 val accuracy: 0.344000
lr 4.200000e-07 reg 3.000000e+04 train accuracy: 0.319796 val accuracy: 0.341000
lr 4.200000e-07 reg 3.250000e+04 train accuracy: 0.311224 val accuracy: 0.327000
lr 4.200000e-07 reg 3.500000e+04 train accuracy: 0.309837 val accuracy: 0.331000
lr 4.200000e-07 reg 3.750000e+04 train accuracy: 0.310367 val accuracy: 0.341000
lr 4.200000e-07 reg 4.000000e+04 train accuracy: 0.316939 val accuracy: 0.334000
lr 4.200000e-07 reg 4.250000e+04 train accuracy: 0.309755 val accuracy: 0.327000
lr 4.200000e-07 reg 4.500000e+04 train accuracy: 0.314898 val accuracy: 0.315000
lr 4.200000e-07 reg 4.750000e+04 train accuracy: 0.293551 val accuracy: 0.310000
lr 4.600000e-07 reg 2.500000e+04 train accuracy: 0.330490 val accuracy: 0.347000
lr 4.600000e-07 reg 2.750000e+04 train accuracy: 0.332204 val accuracy: 0.349000
lr 4.600000e-07 reg 3.000000e+04 train accuracy: 0.318796 val accuracy: 0.333000
lr 4.600000e-07 reg 3.250000e+04 train accuracy: 0.311837 val accuracy: 0.329000
lr 4.600000e-07 reg 3.500000e+04 train accuracy: 0.322918 val accuracy: 0.325000
lr 4.600000e-07 reg 3.750000e+04 train accuracy: 0.297612 val accuracy: 0.314000
lr 4.600000e-07 reg 4.000000e+04 train accuracy: 0.316735 val accuracy: 0.341000
lr 4.600000e-07 reg 4.250000e+04 train accuracy: 0.300755 val accuracy: 0.325000
lr 4.600000e-07 reg 4.500000e+04 train accuracy: 0.305571 val accuracy: 0.330000
lr 4.600000e-07 reg 4.750000e+04 train accuracy: 0.312673 val accuracy: 0.317000
best validation accuracy achieved during cross-validation: 0.355000
```

In [106]:

```
# 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: 0.355000
```

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: A new data point that's placed within the decision boundaries of the SVM such that its can classify that point correctly will add nothing to the loss, because the hinge loss of correctly scored examples is 0. Adding a point to the Softmax classifier's data will increase the loss whether the class is identified correctly or not, because the loss of even correctly classified examples is nonzero if the distribution of scores is not exactly one-hot.

In [107]:

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

```
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])
```

In [ ]:

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])))
```

```
b2 max relative error: 4.447646e-11
W2 max relative error: 3.440708e-09
b1 max relative error: 8.372505e-10
W1 max relative error: 3.561318e-09
```

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

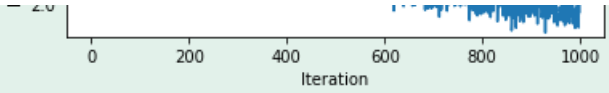
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

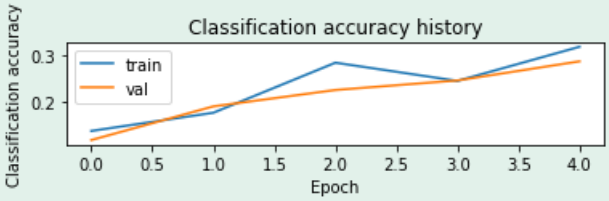
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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]:

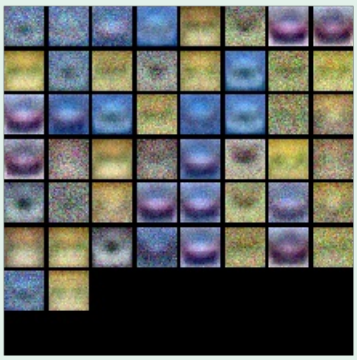
```
# Plot the loss function and train / validation accuracies
plt.subplot(3, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(3, 1, 3)
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 change each parameter a small amount individually, settling on the best one and then freezing it before continuing. If the best parameter is at either end of its tested range, I shifted that range towards the preferred extreme and tried again.

In [75]:

```
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)*****

regularization_strengths = [0.20, 0.30]
results = {}
best_val = -1
input_size = 32 * 32 * 3
num_classes = 10

#ranges
lr_range = np.linspace(2e-4,9e-4,5)
rs_range = np.linspace(regularization_strengths[0],regularization_strengths[1],5)
iters_range = [1500,2000]
hidden_size_range = range(45,65,7)

#initial values
best_lr = lr_range[0]
best_rs = rs_range[0]
best_iters = iters_range[0]
best_hs = hidden_size_range[0]
best_stats = None

#track progress
combinations = len(lr_range) + len(rs_range) + len(iters_range) + len(hidden_size_range)-2
c=1

#sweep lr holding others constant
for lr in lr_range:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_size, best_hs, num_classes)

    stats = net.train(X_train, y_train, X_val, y_val,
        num_iters=best_iters, batch_size=200,
        learning_rate=lr, learning_rate_decay=0.95,
        reg=best_rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(lr,best_rs,best_hs,best_iters)] = (t_accuracy ,v_accuracy, stats)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_lr = lr
        best_stats = stats

#sweep rs holding others constant
for rs in rs_range[1:]:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_size, best_hs, num_classes)

    stats = net.train(X_train, y_train, X_val, y_val,
        num_iters=best_iters, batch_size=200,
        learning_rate=best_lr, learning_rate_decay=0.95,
        reg=rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]
```

```
    v_accuracy = stats["val_acc_history"][-1]

    results[(best_lr,rs,best_hs,best_iters)] = (t_accuracy ,v_accuracy, stats)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_rs = rs
        best_stats = stats

#sweep hidden layer size
for hs in hidden_size_range:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_size, hs, num_classes)

    stats = net.train(X_train, y_train, X_val, y_val,
        num_iters=best_iters, batch_size=200,
        learning_rate=best_lr, learning_rate_decay=0.95,
        reg=best_rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(best_lr,best_rs,hs,best_iters)] = (t_accuracy ,v_accuracy, stats)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_hs = hs
        best_stats = stats

#sweep iters holding others constant
for iters in iters_range[1:]:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_size, best_hs, num_classes)

    stats = net.train(X_train, y_train, X_val, y_val,
        num_iters=iters, batch_size=200,
        learning_rate=best_lr, learning_rate_decay=0.95,
        reg=best_rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(best_lr,best_rs,best_hs,iters)] = (t_accuracy ,v_accuracy, stats)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_iters = iters
        best_stats = stats

# Print out results.
for lr, rs, hidden_size, iters in sorted(results):
    train_accuracy, val_accuracy,stats = results[(lr, rs, hidden_size, iters)]

    print('lr %e reg %e hidden size %d iters %d train accuracy: %f val accuracy: %f' % (
                lr, rs, hidden_size, iters, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
print('best parameters: lr %e rs %e iters %d hidden size %d' % (
                best_lr, best_rs, best_iters, best_hs))

plt.figure()
plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplots_adjust(hspace=0.8)

plt.subplot(2, 1, 2)
plt.plot(best_stats['train_acc_history'], label='train')
```

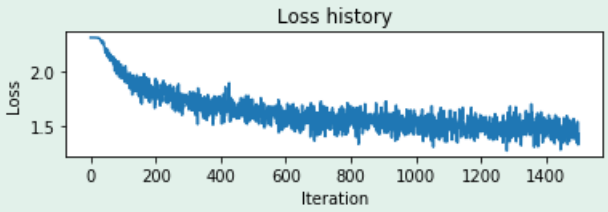
```
plt.plot(best_stats['train_acc_history'], label='train')
plt.plot(best_stats['val_acc_history'], label='val')
plt.title('Classification accuracy history: lr %e rs %e iters %d hidden size %d' % (
                best_lr, best_rs, best_iters, best_hs))
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()

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

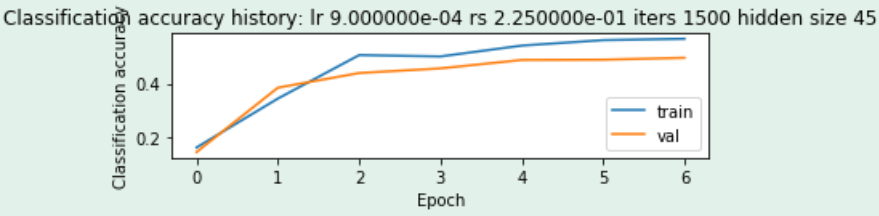
```
Trying combination 1 out of 11 ...
iteration 0 / 1500: loss 2.302874
iteration 100 / 1500: loss 2.299849
iteration 200 / 1500: loss 2.162243
iteration 300 / 1500: loss 2.048810
iteration 400 / 1500: loss 1.931899
iteration 500 / 1500: loss 1.956666
iteration 600 / 1500: loss 1.941663
iteration 700 / 1500: loss 1.884349
iteration 800 / 1500: loss 1.845475
iteration 900 / 1500: loss 1.778048
iteration 1000 / 1500: loss 1.728967
iteration 1100 / 1500: loss 1.700895
iteration 1200 / 1500: loss 1.629444
iteration 1300 / 1500: loss 1.767715
iteration 1400 / 1500: loss 1.627115
Trying combination 2 out of 11 ...
iteration 0 / 1500: loss 2.302845
iteration 100 / 1500: loss 2.225338
iteration 200 / 1500: loss 2.019888
iteration 300 / 1500: loss 1.980237
iteration 400 / 1500: loss 1.835164
iteration 500 / 1500: loss 1.693072
iteration 600 / 1500: loss 1.714224
iteration 700 / 1500: loss 1.730222
iteration 800 / 1500: loss 1.735090
iteration 900 / 1500: loss 1.656751
iteration 1000 / 1500: loss 1.694499
iteration 1100 / 1500: loss 1.674106
iteration 1200 / 1500: loss 1.661362
iteration 1300 / 1500: loss 1.634038
iteration 1400 / 1500: loss 1.612698
Trying combination 3 out of 11 ...
iteration 0 / 1500: loss 2.302838
iteration 100 / 1500: loss 2.095645
iteration 200 / 1500: loss 1.901576
iteration 300 / 1500: loss 1.840796
iteration 400 / 1500: loss 1.833764
iteration 500 / 1500: loss 1.735614
iteration 600 / 1500: loss 1.681332
iteration 700 / 1500: loss 1.576781
iteration 800 / 1500: loss 1.661061
iteration 900 / 1500: loss 1.690657
iteration 1000 / 1500: loss 1.553968
iteration 1100 / 1500: loss 1.689431
iteration 1200 / 1500: loss 1.600033
iteration 1300 / 1500: loss 1.435121
iteration 1400 / 1500: loss 1.391168
Trying combination 4 out of 11 ...
iteration 0 / 1500: loss 2.302860
iteration 100 / 1500: loss 1.986867
iteration 200 / 1500: loss 1.854338
iteration 300 / 1500: loss 1.735244
iteration 400 / 1500: loss 1.605913
iteration 500 / 1500: loss 1.546613
iteration 600 / 1500: loss 1.669641
iteration 700 / 1500: loss 1.592535
iteration 800 / 1500: loss 1.788728
iteration 900 / 1500: loss 1.619000
iteration 1000 / 1500: loss 1.492141
iteration 1100 / 1500: loss 1.466550
iteration 1200 / 1500: loss 1.553445
iteration 1300 / 1500: loss 1.473482
iteration 1400 / 1500: loss 1.459697
Trying combination 5 out of 11 ...
iteration 0 / 1500: loss 2.302853
iteration 100 / 1500: loss 1.982271
```

```
iteration 200 / 1500: loss 1.904947
iteration 300 / 1500: loss 1.779086
iteration 400 / 1500: loss 1.644017
iteration 500 / 1500: loss 1.643210
iteration 600 / 1500: loss 1.714686
iteration 700 / 1500: loss 1.546318
iteration 800 / 1500: loss 1.485663
iteration 900 / 1500: loss 1.588159
iteration 1000 / 1500: loss 1.481466
iteration 1100 / 1500: loss 1.513354
iteration 1200 / 1500: loss 1.439234
iteration 1300 / 1500: loss 1.491573
iteration 1400 / 1500: loss 1.436342
Trying combination 6 out of 11 ...
iteration 0 / 1500: loss 2.302894
iteration 100 / 1500: loss 1.998880
iteration 200 / 1500: loss 1.930920
iteration 300 / 1500: loss 1.699008
iteration 400 / 1500: loss 1.719141
iteration 500 / 1500: loss 1.577062
iteration 600 / 1500: loss 1.569444
iteration 700 / 1500: loss 1.491651
iteration 800 / 1500: loss 1.651888
iteration 900 / 1500: loss 1.616848
iteration 1000 / 1500: loss 1.525363
iteration 1100 / 1500: loss 1.477257
iteration 1200 / 1500: loss 1.500493
iteration 1300 / 1500: loss 1.502172
iteration 1400 / 1500: loss 1.506315
Trying combination 7 out of 11 ...
iteration 0 / 1500: loss 2.302929
iteration 100 / 1500: loss 1.930572
iteration 200 / 1500: loss 1.784247
iteration 300 / 1500: loss 1.679408
iteration 400 / 1500: loss 1.635736
iteration 500 / 1500: loss 1.668294
iteration 600 / 1500: loss 1.575750
iteration 700 / 1500: loss 1.540417
iteration 800 / 1500: loss 1.609449
iteration 900 / 1500: loss 1.570039
iteration 1000 / 1500: loss 1.461331
iteration 1100 / 1500: loss 1.517557
iteration 1200 / 1500: loss 1.495766
iteration 1300 / 1500: loss 1.561595
iteration 1400 / 1500: loss 1.463208
Trying combination 8 out of 11 ...
iteration 0 / 1500: loss 2.302975
iteration 100 / 1500: loss 2.008272
iteration 200 / 1500: loss 1.783955
iteration 300 / 1500: loss 1.685719
iteration 400 / 1500: loss 1.605133
iteration 500 / 1500: loss 1.719082
iteration 600 / 1500: loss 1.506881
iteration 700 / 1500: loss 1.491498
iteration 800 / 1500: loss 1.537859
iteration 900 / 1500: loss 1.601043
iteration 1000 / 1500: loss 1.481936
iteration 1100 / 1500: loss 1.448393
iteration 1200 / 1500: loss 1.538359
iteration 1300 / 1500: loss 1.430175
iteration 1400 / 1500: loss 1.354074
Trying combination 9 out of 11 ...
iteration 0 / 1500: loss 2.302991
iteration 100 / 1500: loss 2.014943
iteration 200 / 1500: loss 1.840861
iteration 300 / 1500: loss 1.595597
iteration 400 / 1500: loss 1.654603
iteration 500 / 1500: loss 1.650839
iteration 600 / 1500: loss 1.600109
iteration 700 / 1500: loss 1.623024
iteration 800 / 1500: loss 1.507264
iteration 900 / 1500: loss 1.569613
iteration 1000 / 1500: loss 1.578438
iteration 1100 / 1500: loss 1.633375
iteration 1200 / 1500: loss 1.698039
iteration 1300 / 1500: loss 1.482963
```

```
iteration 1400 / 1500: loss 1.519255
Trying combination 10 out of 11 ...
iteration 0 / 1500: loss 2.302871
iteration 100 / 1500: loss 1.966722
iteration 200 / 1500: loss 1.803903
iteration 300 / 1500: loss 1.739933
iteration 400 / 1500: loss 1.730545
iteration 500 / 1500: loss 1.646074
iteration 600 / 1500: loss 1.620889
iteration 700 / 1500: loss 1.656620
iteration 800 / 1500: loss 1.586964
iteration 900 / 1500: loss 1.507590
iteration 1000 / 1500: loss 1.525775
iteration 1100 / 1500: loss 1.639454
iteration 1200 / 1500: loss 1.558348
iteration 1300 / 1500: loss 1.513639
iteration 1400 / 1500: loss 1.371905
Trying combination 11 out of 11 ...
iteration 0 / 1500: loss 2.302958
iteration 100 / 1500: loss 1.889937
iteration 200 / 1500: loss 1.746996
iteration 300 / 1500: loss 1.715211
iteration 400 / 1500: loss 1.760376
iteration 500 / 1500: loss 1.563439
iteration 600 / 1500: loss 1.541879
iteration 700 / 1500: loss 1.646364
iteration 800 / 1500: loss 1.478557
iteration 900 / 1500: loss 1.542429
iteration 1000 / 1500: loss 1.527721
iteration 1100 / 1500: loss 1.445080
iteration 1200 / 1500: loss 1.496318
iteration 1300 / 1500: loss 1.629859
iteration 1400 / 1500: loss 1.547904
Trying combination 12 out of 11 ...
iteration 0 / 1500: loss 2.303012
iteration 100 / 1500: loss 2.012889
iteration 200 / 1500: loss 1.843336
iteration 300 / 1500: loss 1.675639
iteration 400 / 1500: loss 1.697953
iteration 500 / 1500: loss 1.554624
iteration 600 / 1500: loss 1.617204
iteration 700 / 1500: loss 1.518619
iteration 800 / 1500: loss 1.647222
iteration 900 / 1500: loss 1.558433
iteration 1000 / 1500: loss 1.632900
iteration 1100 / 1500: loss 1.438842
iteration 1200 / 1500: loss 1.501093
iteration 1300 / 1500: loss 1.602680
iteration 1400 / 1500: loss 1.491014
Trying combination 13 out of 11 ...
iteration 0 / 2000: loss 2.302888
iteration 100 / 2000: loss 1.962113
iteration 200 / 2000: loss 1.855091
iteration 300 / 2000: loss 1.628955
iteration 400 / 2000: loss 1.598208
iteration 500 / 2000: loss 1.542019
iteration 600 / 2000: loss 1.604991
iteration 700 / 2000: loss 1.493482
iteration 800 / 2000: loss 1.598937
iteration 900 / 2000: loss 1.578559
iteration 1000 / 2000: loss 1.672364
iteration 1100 / 2000: loss 1.534014
iteration 1200 / 2000: loss 1.379586
iteration 1300 / 2000: loss 1.527801
iteration 1400 / 2000: loss 1.307030
iteration 1500 / 2000: loss 1.520774
iteration 1600 / 2000: loss 1.520051
iteration 1700 / 2000: loss 1.560118
iteration 1800 / 2000: loss 1.383713
iteration 1900 / 2000: loss 1.431190
lr 2.000000e-04 reg 2.000000e-01 hidden size 45 iters 1500 train accuracy: 0.455000 val accuracy:
0.403000
lr 3.750000e-04 reg 2.000000e-01 hidden size 45 iters 1500 train accuracy: 0.445000 val accuracy:
0.444000
lr 5.500000e-04 reg 2.000000e-01 hidden size 45 iters 1500 train accuracy: 0.515000 val accuracy:
0.463000
lr 7.250000e-04 reg 2.000000e-01 hidden size 45 iters 1500 train accuracy: 0.580000 val accuracy:
0.473000
```

```
lr 9.000000e-04 reg 2.000000e-01 hidden size 45 iters 1500 train accuracy: 0.560000 val accuracy:
0.480000
lr 9.000000e-04 reg 2.250000e-01 hidden size 45 iters 1500 train accuracy: 0.555000 val accuracy:
0.468000
lr 9.000000e-04 reg 2.250000e-01 hidden size 45 iters 2000 train accuracy: 0.605000 val accuracy:
0.483000
lr 9.000000e-04 reg 2.250000e-01 hidden size 52 iters 1500 train accuracy: 0.560000 val accuracy:
0.493000
lr 9.000000e-04 reg 2.250000e-01 hidden size 59 iters 1500 train accuracy: 0.510000 val accuracy:
0.475000
lr 9.000000e-04 reg 2.500000e-01 hidden size 45 iters 1500 train accuracy: 0.585000 val accuracy:
0.490000
lr 9.000000e-04 reg 2.750000e-01 hidden size 45 iters 1500 train accuracy: 0.560000 val accuracy:
0.488000
lr 9.000000e-04 reg 3.000000e-01 hidden size 45 iters 1500 train accuracy: 0.595000 val accuracy:
0.478000
best validation accuracy achieved during cross-validation: 0.495000
best parameters: lr 9.000000e-04 rs 2.250000e-01 iters 1500 hidden size 45
```





In [76]:

```
# 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 [77]:

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

```
Test accuracy:  0.473
```

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. 3.

YourExplanation:

1- Training on a larger dataset will increase the generalizability of our model by increasing the representativeness of the training data. This isn't always feasible, though.

3- A larger regularization strength will smooth the fitting by constraining the size of the parameters further.

2 would be a mistake- Adding hidden units would allow the decision boundaries to become more complex, actually increasing overfitting.

In [ ]:

```
```

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.

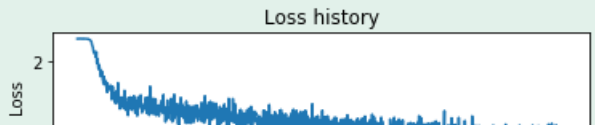
In [74]:

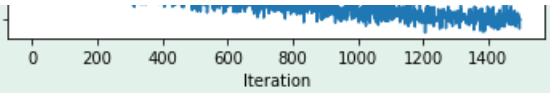
```
# Use the validation set to tune the learning rate and regularization strength

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)*****

best_lr=None
best_rs=None
for lr in np.linspace(1e-9,5e-9,5):
    for rs in np.logspace(5,8,5):
        svm = LinearSVM()
        loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr, reg=rs,
                              num_iters=500, verbose=True)

        Yt = svm.predict(X_train_feats)
        Yp = svm.predict(X_val_feats)

        t_num_correct = np.sum(Yt == y_train)
        t_accuracy = float(t_num_correct) / X_train.shape[0]

        v_num_correct = np.sum(Yp == y_val)
        v_accuracy = float(v_num_correct) / X_val.shape[0]

        results[(lr,rs)] = (t_accuracy ,v_accuracy)

        if (best_val<v_accuracy):
            best_val = v_accuracy
            best_svm = svm
            best_lr=lr
            best_rs=rs

plt.figure()
plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
```

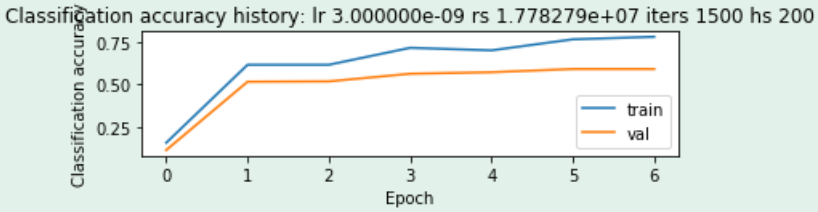
```
plt.ylabel('Loss')

plt.subplots_adjust(hspace=0.8)

plt.subplot(2, 1, 2)
plt.plot(best_stats['train_acc_history'], label='train')
plt.plot(best_stats['val_acc_history'], label='val')
plt.title('Classification accuracy history: lr %e rs %e iters %d hs %d' % (
                best_lr, best_rs, best_iters, best_hs))
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()

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

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

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

```
iteration 0 / 500: loss 165.325536
iteration 100 / 500: loss 159.201907
iteration 200 / 500: loss 153.309343
iteration 300 / 500: loss 147.665957
iteration 400 / 500: loss 142.226601
iteration 0 / 500: loss 863.581588
iteration 100 / 500: loss 691.358649
iteration 200 / 500: loss 553.825219
iteration 300 / 500: loss 444.029965
iteration 400 / 500: loss 356.351636
iteration 0 / 500: loss 4973.710939
iteration 100 / 500: loss 1404.743995
iteration 200 / 500: loss 401.386823
iteration 300 / 500: loss 119.312513
iteration 400 / 500: loss 40.011851
iteration 0 / 500: loss 28060.418298
iteration 100 / 500: loss 29.068165
iteration 200 / 500: loss 9.014355
iteration 300 / 500: loss 9.000009
iteration 400 / 500: loss 8.999999
iteration 0 / 500: loss 155402.286807
iteration 100 / 500: loss 9.000000
iteration 200 / 500: loss 9.000000
iteration 300 / 500: loss 9.000000
iteration 400 / 500: loss 9.000000
iteration 0 / 500: loss 168.451153
iteration 100 / 500: loss 156.163319
iteration 200 / 500: loss 144.846800
iteration 300 / 500: loss 134.387967
iteration 400 / 500: loss 124.754438
iteration 0 / 500: loss 894.592233
iteration 100 / 500: loss 573.452654
iteration 200 / 500: loss 368.768691
iteration 300 / 500: loss 238.317409
iteration 400 / 500: loss 155.161049
iteration 0 / 500: loss 4756.976960
iteration 100 / 500: loss 381.229513
iteration 200 / 500: loss 38.182642
iteration 300 / 500: loss 11.288025
iteration 400 / 500: loss 9.179358
iteration 0 / 500: loss 27354.686245
iteration 100 / 500: loss 9.010667
iteration 200 / 500: loss 8.999999
iteration 300 / 500: loss 8.999999
iteration 400 / 500: loss 8.999999
iteration 0 / 500: loss 157374.845036
iteration 100 / 500: loss 9.000000
iteration 200 / 500: loss 9.000000
iteration 300 / 500: loss 9.000000
iteration 400 / 500: loss 9.000000
iteration 0 / 500: loss 165.597579
iteration 100 / 500: loss 147.876830
iteration 200 / 500: loss 132.174216
```

```
iteration 300 / 500: loss 118.240150
iteration 400 / 500: loss 105.889545
iteration 0 / 500: loss 861.202960
iteration 100 / 500: loss 442.476153
iteration 200 / 500: loss 229.509130
iteration 300 / 500: loss 121.165903
iteration 400 / 500: loss 66.055464
iteration 0 / 500: loss 5150.527544
iteration 100 / 500: loss 120.486508
iteration 200 / 500: loss 11.417583
iteration 300 / 500: loss 9.052500
iteration 400 / 500: loss 9.001130
iteration 0 / 500: loss 26707.599126
iteration 100 / 500: loss 9.000003
iteration 200 / 500: loss 8.999999
iteration 300 / 500: loss 8.999999
iteration 400 / 500: loss 8.999999
iteration 0 / 500: loss 148269.771980
iteration 100 / 500: loss 9.000000
iteration 200 / 500: loss 9.000000
iteration 300 / 500: loss 9.000000
iteration 400 / 500: loss 9.000000
iteration 0 / 500: loss 166.094156
iteration 100 / 500: loss 142.856297
iteration 200 / 500: loss 123.051278
iteration 300 / 500: loss 106.186850
iteration 400 / 500: loss 91.834432
iteration 0 / 500: loss 945.609671
iteration 100 / 500: loss 389.127031
iteration 200 / 500: loss 163.272839
iteration 300 / 500: loss 71.614498
iteration 400 / 500: loss 34.411979
iteration 0 / 500: loss 4744.897374
iteration 100 / 500: loss 37.167659
iteration 200 / 500: loss 9.167549
iteration 300 / 500: loss 9.000989
iteration 400 / 500: loss 9.000001
iteration 0 / 500: loss 27410.913860
iteration 100 / 500: loss 8.999999
iteration 200 / 500: loss 8.999999
iteration 300 / 500: loss 8.999999
iteration 400 / 500: loss 8.999999
iteration 0 / 500: loss 161306.941662
iteration 100 / 500: loss 9.000000
iteration 200 / 500: loss 9.000000
iteration 300 / 500: loss 9.000000
iteration 400 / 500: loss 9.000000
iteration 0 / 500: loss 157.410758
iteration 100 / 500: loss 130.491878
iteration 200 / 500: loss 108.460678
iteration 300 / 500: loss 90.412024
iteration 400 / 500: loss 75.650674
iteration 0 / 500: loss 811.416324
iteration 100 / 500: loss 268.768968
iteration 200 / 500: loss 93.096630
iteration 300 / 500: loss 36.224473
iteration 400 / 500: loss 17.811652
iteration 0 / 500: loss 5106.776765
iteration 100 / 500: loss 17.246844
iteration 200 / 500: loss 9.013323
iteration 300 / 500: loss 9.000017
iteration 400 / 500: loss 8.999995
iteration 0 / 500: loss 26319.747543
iteration 100 / 500: loss 8.999999
iteration 200 / 500: loss 8.999999
iteration 300 / 500: loss 8.999999
iteration 400 / 500: loss 8.999999
iteration 0 / 500: loss 150193.743918
iteration 100 / 500: loss 9.000000
iteration 200 / 500: loss 9.000000
iteration 300 / 500: loss 9.000000
iteration 400 / 500: loss 9.000000
```








```
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.072265 val accuracy: 0.072000
lr 1.000000e-09 reg 5.623413e+05 train accuracy: 0.101041 val accuracy: 0.100000
lr 1.000000e-09 reg 3.162278e+06 train accuracy: 0.086429 val accuracy: 0.099000
lr 1.000000e-09 reg 1.778279e+07 train accuracy: 0.408837 val accuracy: 0.407000
lr 1.000000e-09 reg 1.000000e+08 train accuracy: 0.409429 val accuracy: 0.407000
lr 2.000000e-09 reg 1.000000e+05 train accuracy: 0.098878 val accuracy: 0.124000
lr 2.000000e-09 reg 5.623413e+05 train accuracy: 0.114469 val accuracy: 0.111000
lr 2.000000e-09 reg 3.162278e+06 train accuracy: 0.109224 val accuracy: 0.114000
lr 2.000000e-09 reg 1.778279e+07 train accuracy: 0.412959 val accuracy: 0.399000
lr 2.000000e-09 reg 1.000000e+08 train accuracy: 0.394102 val accuracy: 0.383000
lr 3.000000e-09 reg 1.000000e+05 train accuracy: 0.116592 val accuracy: 0.127000
lr 3.000000e-09 reg 5.623413e+05 train accuracy: 0.098571 val accuracy: 0.098000
lr 3.000000e-09 reg 3.162278e+06 train accuracy: 0.292490 val accuracy: 0.290000
lr 3.000000e-09 reg 1.778279e+07 train accuracy: 0.412714 val accuracy: 0.440000
lr 3.000000e-09 reg 1.000000e+08 train accuracy: 0.370449 val accuracy: 0.373000
lr 4.000000e-09 reg 1.000000e+05 train accuracy: 0.089408 val accuracy: 0.084000
lr 4.000000e-09 reg 5.623413e+05 train accuracy: 0.109980 val accuracy: 0.096000
lr 4.000000e-09 reg 3.162278e+06 train accuracy: 0.410633 val accuracy: 0.412000
lr 4.000000e-09 reg 1.778279e+07 train accuracy: 0.407306 val accuracy: 0.401000
lr 4.000000e-09 reg 1.000000e+08 train accuracy: 0.348163 val accuracy: 0.334000
lr 5.000000e-09 reg 1.000000e+05 train accuracy: 0.111612 val accuracy: 0.108000
lr 5.000000e-09 reg 5.623413e+05 train accuracy: 0.102816 val accuracy: 0.105000
lr 5.000000e-09 reg 3.162278e+06 train accuracy: 0.412918 val accuracy: 0.417000
lr 5.000000e-09 reg 1.778279e+07 train accuracy: 0.389286 val accuracy: 0.379000
lr 5.000000e-09 reg 1.000000e+08 train accuracy: 0.344694 val accuracy: 0.331000
best validation accuracy achieved during cross-validation: 0.440000
```

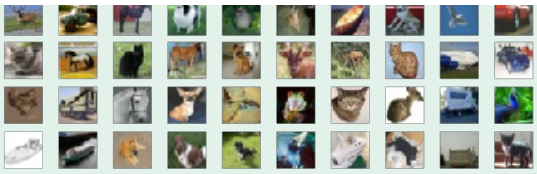
In [75]:

```
# 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)
```

```
0.414
```

In [76]:

```
# 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:

It seems like most failures are pictures of animals with very close framing. This makes sense if the classifier mostly learned the profile of the animals without focusing on features like their fine details or fur patterns. At this low resolution, it seems likely that the driving force behind many of these is the background color/uniformity. For example, most images that were misclassified as ships have blue backgrounds.

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 [72]:

```
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)*****

regularization_strengths = [1e-5, 1e-4]
results = {}
best_val = -1
input_size = 32 * 32 * 3
num_classes = 10

#ranges
lr_range = np.linspace(1e-1,5e-1,5)
rs_range = np.linspace(regularization_strengths[0],regularization_strengths[1],4)
iters_range = [1500]
hs_range = np.linspace(200,400,4, dtype=int)

#initial values
best_lr = lr_range[0]
best_rs = rs_range[0]
best_iters = iters_range[0]
best_stats=None
best_hs=hs_range[0]

#track progress
combinations = len(lr_range) + len(rs_range) + len(iters_range) +len(hs_range) -2
c=1

#sweep lr holding others constant
```

```
#sweep lr holding others constant
for lr in lr_range:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_dim, hidden_dim, num_classes)
    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
        num_iters=best_iters, batch_size=200,
        learning_rate=lr, learning_rate_decay=0.95,
        reg=best_rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(lr,best_rs,best_iters)] = (t_accuracy ,v_accuracy)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_lr = lr
        best_stats = stats

#sweep rs holding others constant
for rs in rs_range[1:]:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_dim, hidden_dim, num_classes)

    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
        num_iters=best_iters, batch_size=200,
        learning_rate=best_lr, learning_rate_decay=0.95,
        reg=rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(best_lr,rs,best_iters)] = (t_accuracy ,v_accuracy)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_rs = rs
        best_stats = stats

#sweep iters holding others constant
for iters in iters_range[1:]:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_dim, hidden_dim, num_classes)

    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
        num_iters=iters, batch_size=200,
        learning_rate=best_lr, learning_rate_decay=0.95,
        reg=best_rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(best_lr,best_rs,iters)] = (t_accuracy ,v_accuracy)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_iters = iters
        best_stats = stats

for hs in hs_range[1:]:
    print("Trying combination",c,"out of",combinations,"...")
    c+=1

    net = TwoLayerNet(input_dim, hs, num_classes)

    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
        num_iters=best_iters, batch_size=200,
        learning_rate=best_lr, learning_rate_decay=0.95,
        reg=best_rs, verbose=True)
```

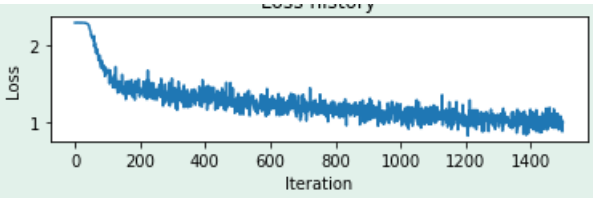
```
        reg=best_rs, verbose=True)

    t_accuracy = stats["train_acc_history"][-1]
    v_accuracy = stats["val_acc_history"][-1]

    results[(best_lr,best_rs,best_iters)] = (t_accuracy ,v_accuracy)

    if (best_val<v_accuracy):
        best_val = v_accuracy
        best_net = net
        best_hs = hs
        best_stats = stats

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

print('best validation accuracy achieved during cross-validation: %f' % best_val)
print('best parameters: lr %e rs %e iters %d hs %d' % (
                best_lr, best_rs, best_iters, best_hs))

plt.figure()
plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplots_adjust(hspace=0.8)

plt.subplot(2, 1, 2)
plt.plot(best_stats['train_acc_history'], label='train')
plt.plot(best_stats['val_acc_history'], label='val')
plt.title('Classification accuracy history: lr %e rs %e iters %d hs %d' % (
                best_lr, best_rs, best_iters, best_hs))
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()

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

```
Trying combination 1 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 2.302196
iteration 200 / 1500: loss 2.151353
iteration 300 / 1500: loss 1.779974
iteration 400 / 1500: loss 1.643599
iteration 500 / 1500: loss 1.477532
iteration 600 / 1500: loss 1.395102
iteration 700 / 1500: loss 1.478518
iteration 800 / 1500: loss 1.316066
iteration 900 / 1500: loss 1.319829
iteration 1000 / 1500: loss 1.439714
iteration 1100 / 1500: loss 1.190621
iteration 1200 / 1500: loss 1.285246
iteration 1300 / 1500: loss 1.424394
iteration 1400 / 1500: loss 1.234824
Trying combination 2 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 2.142591
iteration 200 / 1500: loss 1.650439
iteration 300 / 1500: loss 1.436816
iteration 400 / 1500: loss 1.508764
iteration 500 / 1500: loss 1.416384
iteration 600 / 1500: loss 1.420158
iteration 700 / 1500: loss 1.211508
iteration 800 / 1500: loss 1.423228
iteration 900 / 1500: loss 1.235630
iteration 1000 / 1500: loss 1.172756
iteration 1100 / 1500: loss 1.273249
iteration 1200 / 1500: loss 1.171010
iteration 1300 / 1500: loss 1.068868
iteration 1400 / 1500: loss 1.198778
Trying combination 3 out of 12 ...
iteration 0 / 1500: loss 2.302585
```

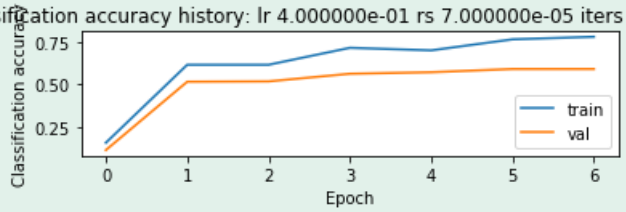
```
iteration 100 / 1500: loss 1.806373
iteration 200 / 1500: loss 1.433350
iteration 300 / 1500: loss 1.332330
iteration 400 / 1500: loss 1.196744
iteration 500 / 1500: loss 1.342577
iteration 600 / 1500: loss 1.399055
iteration 700 / 1500: loss 1.138922
iteration 800 / 1500: loss 1.160413
iteration 900 / 1500: loss 1.192858
iteration 1000 / 1500: loss 1.124036
iteration 1100 / 1500: loss 1.038154
iteration 1200 / 1500: loss 1.216531
iteration 1300 / 1500: loss 1.053608
iteration 1400 / 1500: loss 1.062656
Trying combination 4 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.668172
iteration 200 / 1500: loss 1.337925
iteration 300 / 1500: loss 1.326293
iteration 400 / 1500: loss 1.393438
iteration 500 / 1500: loss 1.148164
iteration 600 / 1500: loss 1.228272
iteration 700 / 1500: loss 1.122720
iteration 800 / 1500: loss 1.210196
iteration 900 / 1500: loss 1.236802
iteration 1000 / 1500: loss 1.076814
iteration 1100 / 1500: loss 1.044808
iteration 1200 / 1500: loss 1.074283
iteration 1300 / 1500: loss 1.008359
iteration 1400 / 1500: loss 0.994109
Trying combination 5 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.412171
iteration 200 / 1500: loss 1.292677
iteration 300 / 1500: loss 1.293791
iteration 400 / 1500: loss 1.375756
iteration 500 / 1500: loss 1.202977
iteration 600 / 1500: loss 1.234561
iteration 700 / 1500: loss 1.046254
iteration 800 / 1500: loss 1.053739
iteration 900 / 1500: loss 1.095995
iteration 1000 / 1500: loss 1.074169
iteration 1100 / 1500: loss 0.968738
iteration 1200 / 1500: loss 1.018769
iteration 1300 / 1500: loss 0.926519
iteration 1400 / 1500: loss 1.006342
Trying combination 6 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.594631
iteration 200 / 1500: loss 1.353696
iteration 300 / 1500: loss 1.230202
iteration 400 / 1500: loss 1.350463
iteration 500 / 1500: loss 1.135931
iteration 600 / 1500: loss 1.118644
iteration 700 / 1500: loss 1.210996
iteration 800 / 1500: loss 1.186099
iteration 900 / 1500: loss 1.213139
iteration 1000 / 1500: loss 1.103100
iteration 1100 / 1500: loss 1.062984
iteration 1200 / 1500: loss 1.156172
iteration 1300 / 1500: loss 0.938886
iteration 1400 / 1500: loss 0.940454
Trying combination 7 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.643702
iteration 200 / 1500: loss 1.398352
iteration 300 / 1500: loss 1.336526
iteration 400 / 1500: loss 1.302256
iteration 500 / 1500: loss 1.147226
iteration 600 / 1500: loss 1.202920
iteration 700 / 1500: loss 1.181080
iteration 800 / 1500: loss 1.098679
iteration 900 / 1500: loss 1.022442
iteration 1000 / 1500: loss 1.231443
iteration 1100 / 1500: loss 1.066671
iteration 1200 / 1500: loss 0.990274
iteration 1300 / 1500: loss 1.007600
```

```
iteration 1500 / 1500: loss 1.007000
iteration 1400 / 1500: loss 0.987474
Trying combination 8 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.747404
iteration 200 / 1500: loss 1.343882
iteration 300 / 1500: loss 1.423145
iteration 400 / 1500: loss 1.272587
iteration 500 / 1500: loss 1.197517
iteration 600 / 1500: loss 1.246355
iteration 700 / 1500: loss 1.176174
iteration 800 / 1500: loss 1.176263
iteration 900 / 1500: loss 1.096395
iteration 1000 / 1500: loss 1.005182
iteration 1100 / 1500: loss 1.200498
iteration 1200 / 1500: loss 1.082977
iteration 1300 / 1500: loss 1.062186
iteration 1400 / 1500: loss 1.148685
Trying combination 9 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.672426
iteration 200 / 1500: loss 1.386822
iteration 300 / 1500: loss 1.414621
iteration 400 / 1500: loss 1.220301
iteration 500 / 1500: loss 1.266055
iteration 600 / 1500: loss 1.171636
iteration 700 / 1500: loss 1.221490
iteration 800 / 1500: loss 1.082512
iteration 900 / 1500: loss 1.230266
iteration 1000 / 1500: loss 1.082942
iteration 1100 / 1500: loss 1.154237
iteration 1200 / 1500: loss 1.054577
iteration 1300 / 1500: loss 0.996363
iteration 1400 / 1500: loss 0.895911
Trying combination 10 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.833209
iteration 200 / 1500: loss 1.430301
iteration 300 / 1500: loss 1.333443
iteration 400 / 1500: loss 1.279536
iteration 500 / 1500: loss 1.277317
iteration 600 / 1500: loss 1.324206
iteration 700 / 1500: loss 1.128354
iteration 800 / 1500: loss 1.082774
iteration 900 / 1500: loss 1.134261
iteration 1000 / 1500: loss 1.042566
iteration 1100 / 1500: loss 1.093032
iteration 1200 / 1500: loss 1.328485
iteration 1300 / 1500: loss 1.084404
iteration 1400 / 1500: loss 0.933869
Trying combination 11 out of 12 ...
iteration 0 / 1500: loss 2.302585
iteration 100 / 1500: loss 1.687438
iteration 200 / 1500: loss 1.429971
iteration 300 / 1500: loss 1.395175
iteration 400 / 1500: loss 1.347003
iteration 500 / 1500: loss 1.255599
iteration 600 / 1500: loss 1.191490
iteration 700 / 1500: loss 1.286219
iteration 800 / 1500: loss 1.222003
iteration 900 / 1500: loss 1.177781
iteration 1000 / 1500: loss 1.246843
iteration 1100 / 1500: loss 1.069009
iteration 1200 / 1500: loss 1.198427
iteration 1300 / 1500: loss 0.947747
iteration 1400 / 1500: loss 0.889320
lr 1.000000e-01 rs 1.000000e-05 iters 1500 train accuracy: 0.585000 val accuracy: 0.534000
lr 2.000000e-01 rs 1.000000e-05 iters 1500 train accuracy: 0.710000 val accuracy: 0.563000
lr 3.000000e-01 rs 1.000000e-05 iters 1500 train accuracy: 0.740000 val accuracy: 0.567000
lr 4.000000e-01 rs 1.000000e-05 iters 1500 train accuracy: 0.805000 val accuracy: 0.581000
lr 4.000000e-01 rs 4.000000e-05 iters 1500 train accuracy: 0.780000 val accuracy: 0.582000
lr 4.000000e-01 rs 7.000000e-05 iters 1500 train accuracy: 0.780000 val accuracy: 0.562000
lr 4.000000e-01 rs 1.000000e-04 iters 1500 train accuracy: 0.820000 val accuracy: 0.588000
lr 5.000000e-01 rs 1.000000e-05 iters 1500 train accuracy: 0.895000 val accuracy: 0.574000
best validation accuracy achieved during cross-validation: 0.590000
best parameters: lr 4.000000e-01 rs 7.000000e-05 iters 1500 hs 200
```

Loss history



Classification accuracy history: lr 4.000000e-01 rs 7.000000e-05 iters 1500 hs 200



In [73]:

```
# 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)
```

```
0.579
```

In [ ]:

```

```

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(self.X_train-X[i,:]), 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)*****

        # X_tile = np.expand_dims(X, axis=1)
        # print('tiled the training data:', X_tile.shape)

        # X_diffs_sq = np.square(X_tile - self.X_train)
        # print('calculated pixel diffs and squared', X_diffs_sq.shape)

        # X_diff_sq_sum_sqrt = np.sqrt(np.sum(X_diffs_sq, axis=2))
        # print('eliminated long axis by summing, then took sqrt', X_diff_sq_sum_sqrt.shape)

        term1 = np.sum(np.square(np.expand_dims(X, axis=1)), axis=2)
        term2 = -2*X@self.X_train.T
        term3 = np.sum(np.square(self.X_train), axis=1)

        dists = np.sqrt(  term1 + term2 + term3 )

        # *****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)*****

            idx = np.argsort(dists[i,:])[0:k]
            closest_y = self.y_train[idx]

            # *****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)*****

            c_y_unique = np.unique(closest_y, return_counts=True);
            closest_y_idx = np.argmax(c_y_unique[1]);
            y_pred[i] = c_y_unique[0][closest_y_idx]

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

        return y_pred
```

2 linear_classifier.py

```python
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)*****

            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)*****

            # 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 += - grad * 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))

        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)*****

        scores = X@self.W
        y_pred = np.argmax(scores, 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

```
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
                dW[:,j] += X[i]
                dW[:,y[i]] += -X[i]

    # 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)*****

    #update gradient for the extra term and the scaling
    dW /= num_train
    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)*****
    N_train = X.shape[0] #500

    #get every score
    scores = X@W

    #get correct class' scores
    correct_scores = scores[np.arange(N_train),y]

    #calculate all scores' diff from correct score (including the correct score w/ itself)
    score_diffs = scores - np.expand_dims(correct_scores, -1) + 1 #add delta=1

    #take max(0,margin)
    score_diffs_capped = np.maximum(np.zeros(score_diffs.shape),score_diffs)

    #sum up score diffs by row, subtract 1 for the mean(0,1) that resulted in each row from calculating a
    diff for the correct score vs itself
    Loss_i = np.sum(score_diffs_capped, axis=1) - 1

    #normalize and add regularization term
    loss = np.sum(Loss_i, axis=0) / N_train + 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)*****

    #start with the capped margins from earlier to use as multiples of X
    dw_col = score_diffs_capped

    #theres a 1 for ever term where the margin was high enough that we added Loss (including 1 unit of
    loss for correctxcorrect locations)
    dw_col[score_diffs_capped>0] = 1 #representative of the result of the "if margin>0" above
    times_contributed = np.sum(dw_col, axis=1)

    #at correct class locations the contribution to loss is actually -X so put a -1*(for each time one of
    the non-correct classes subtracted it, so times_contributed-1)
    dw_col[np.arange(N_train),y] = -(times_contributed-1)

    #add X to the gradient wherever the multiple was positive, add -X an appropriate # of times in the
    correct class places
    dW = X.T@dw_col

    #update gradient for the extra term and the scaling
    dW /= N_train
    dW += 2*reg*W


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

    return loss, dW
```

4 softmax.py

```python
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)*****

    # compute the loss and the gradient (from SVM)
    num_classes = W.shape[1]
    num_train = X.shape[0]

    for i in range(num_train):
      scores = X[i].dot(W)
      #use logC
      scores -= np.max(scores)
      correct_class_score = scores[y[i]]

      #begin counting up L_i and denominator summation
      l_sub_i = -correct_class_score
      sum_total = 0

      #calculate summation
      for j in range(num_classes):
          sum_total += np.exp(scores[j])

      #add summation to L_i
      l_sub_i += np.log(sum_total)

      #add L_i to overall loss
      loss += l_sub_i

      #add terms to dW for weights used in summation and for correct class weights used
      #we need to loop again because we needed to know the summation total for this
      #print(W.shape, X.shape)
      dW[:,y[i]] += -X[i]
      for j in range(num_classes):
        dW[:,j] += (1/sum_total) * X[i] * np.exp(scores[j])

    # 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
    dW /= num_train

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

```
    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_train = X.shape[0]

    #get all scores via matrix multiplication
    scores = X.dot(W)
    #normalization trick to avoid instability
    scores -= np.expand_dims(np.max(scores, axis=1),-1)

    #start per example loss as -correct class score
    correct_class_scores = scores[np.arange(num_train),y]
    L_sub_is = -correct_class_scores

    #perform the summation using np.sum
    summation_totals = np.sum(np.exp(scores), axis=1)

    #add contributions of the summations to the gradient
    dW_per_example = (np.exp(scores)*np.expand_dims(1/summation_totals,-1))
    dW_per_example[np.arange(num_train),y] -= 1 #to give a -X contribution for every correct class
    dW += X.T.dot(dW_per_example)

    #add summation totals to the per example losses
    L_sub_is += np.log(summation_totals)

    #add up losses per example
    loss = np.sum(L_sub_is)

    #take avg and add regularization loss
    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
```

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)*****

        hidden_layer_preactiv = X.dot(W1) + b1
        hidden_layer = np.maximum(np.zeros_like(hidden_layer_preactiv), hidden_layer_preactiv)
        scores = hidden_layer.dot(W2) + b2

        # *****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)*****

        #normalization trick to avoid instability
        scores -= np.expand_dims(np.max(scores, axis=1),-1)

        #start per example loss as -correct class score
        correct_class_scores = scores[np.arange(N),y]
        L_sub_is = -correct_class_scores

        #perform the summation using np.sum
        summation_totals = np.sum(np.exp(scores), axis=1)

        #add summation totals to the per example losses
        L_sub_is += np.log(summation_totals)

        #add up losses per example
        loss = np.sum(L_sub_is)

        #take avg and add regularization loss
        loss /= N
        loss += reg * np.sum(W1 * W1)
        loss += reg * 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)*****

        #calculate gradient of all sub losses wrt scores
        dLi_ds = (np.exp(scores)*np.expand_dims(1/summation_totals,-1))
        dLi_ds[np.arange(N),y] -= 1 #to give a -1 contribution for every correct class
        dL_ds = dLi_ds #no score contributes to Li for an other example (another i) so dL/ds = dLi/ds
        dL_ds = (dL_ds / N) #take avg effect not sum

        #calculate grad of loss wrt b2 --avoid calculating jacobian ds_db2, just write implicitly!
        dL_db2 = np.sum(dL_ds, axis=0)
        grads['b2'] = dL_db2

        #calculate grad of loss wrt W2
        #ds/dW2 = hidden_layer
        dL_dw2 = hidden_layer.T @ dL_ds + 2 * reg * W2 #also add grad loss due to reg
        grads['W2'] = dL_dw2

```

```
        #calculate grad of loss wrt hidden_layer
        #ds/dhl = w2
        dL_dhl = dL_ds @ W2.T

        #calculate grad of loss wrt hidden_layer_preactiv
        dL_dhlp = dL_dhl
        dL_dhlp[hidden_layer_preactiv < 0] = 0

        #calculate grad of loss wrt b1
        dL_db1 = np.sum(dL_dhlp, axis=0)
        grads['b1'] = dL_db1

        #calculate grad of loss wrt W1
        dL_dw1 = X.T @ dL_dhlp + 2 * reg * W1 #also add grad loss due to reg
        grads['W1'] = dL_dw1

        # *****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)*****

            mask = np.random.choice(np.arange(X.shape[0]), (batch_size))
            X_batch = X[mask,:]
            y_batch = y[mask]

            # *****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)*****
```

```
            for param_name in grads:
              self.params[param_name] += -grads[param_name] * 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)*****

        scores = self.loss(X)
        y_pred = np.argmax(scores, axis=1)

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

        return y_pred
```