k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

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

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

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

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

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

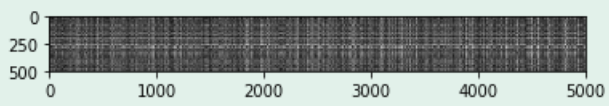
In [6]:

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

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

```
(500, 5000)
```

In [7]:

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



Inline Question 1

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

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

YourAnswer: The bright rows are caused by test images that are distant from all training examples. Conversely, the bright columns represent training images that are very different from all test examples.

In [8]:

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

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

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

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

In [9]:

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

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply.

1. Subtracting the mean μ ($\tilde{P}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{P}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data.

YourAnswer: 1,3,5

YourExplanation: Preprocessing operations 1,3,5 all modify each of the pixel data by some constant operator, either through multiplication / division or by rotating the coordinate frame. Thus, even though the absolute pixel values are changed, their relative values assessed on a linear basis (i.e. the L1 distance) will not be different.

```
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 32.064571 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 [8]:

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

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

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

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


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Iterate through each value of k (for knn)
for k in k_choices:
    # Create array to store fold accuracies for each k-value
    k_accs = []
    # Run over each fold (for train/test split cross-validation)
    for i in range(num_folds):
        # Take ith fold as validation set
        xVal = np.array(X_train_folds[i])
        yVal = np.array(y_train_folds[i])

        # Concatenate together all other (i-1) folds which are not the ith fold
        xTrain = np.vstack(X_train_folds[:i] + X_train_folds[i+1:])
        yTrain = np.vstack(y_train_folds[:i] + y_train_folds[i+1:])
        yTrain = yTrain.reshape((-1,))
#         xTrainFolds = [X_train_folds[i] for k in range(num_folds) if k != i]
#         xTrain = np.concatenate(xTrainFolds)
#         yTrainFolds = [y_train_folds[i] for k in range(num_folds) if k != i]
#         yTrain = np.concatenate(yTrainFolds)

        # Train our classifier
        classifier = KNearestNeighbor()
        classifier.train(xTrain, yTrain)

        # Test against the validation set
        yPred = classifier.predict(xVal, k=k)

        # Match against validation labels and calculate accuracy
        results = np.sum(yPred == yVal)
        nVal = yVal.shape[0]
        accuracy = results/nVal

        k_accs.append(accuracy)

    # Store the list of accuracies for each fold into the dict
    k_to_accuracies[k] = k_accs
```

```
    pass

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

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

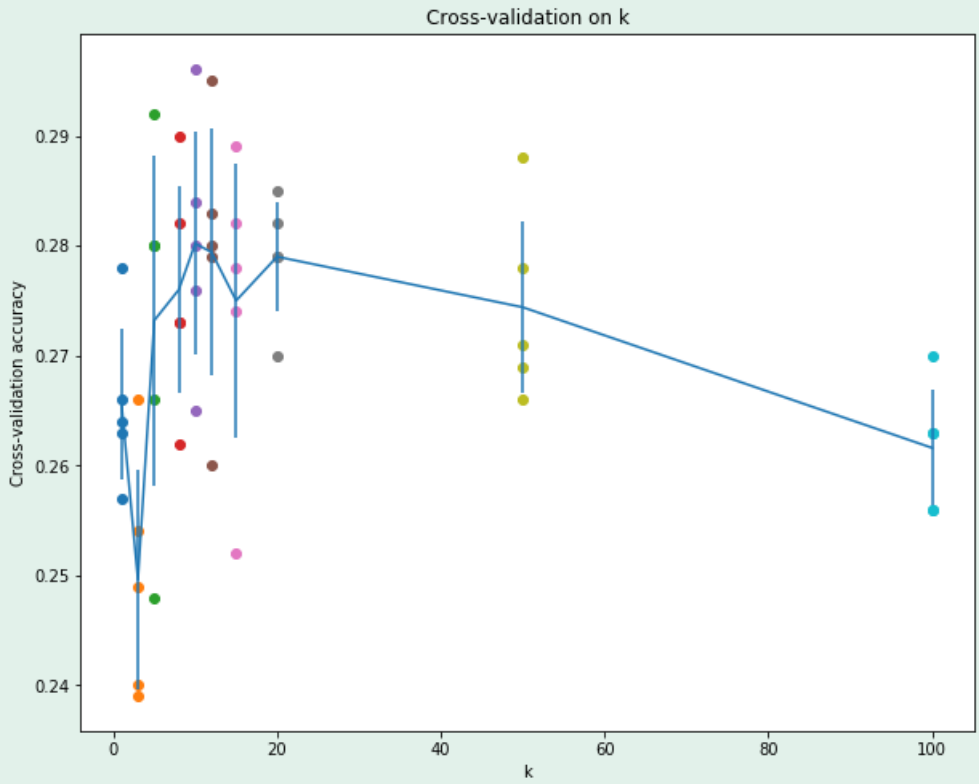
```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000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

In [11]:

```
k_to_accuracies
```

Out[11]:

```
{1: [0.263, 0.257, 0.264, 0.278, 0.266],
 3: [0.239, 0.249, 0.24, 0.266, 0.254],
 5: [0.248, 0.266, 0.28, 0.292, 0.28],
 8: [0.262, 0.282, 0.273, 0.29, 0.273],
 10: [0.265, 0.296, 0.276, 0.284, 0.28],
 12: [0.26, 0.295, 0.279, 0.283, 0.28],
 15: [0.252, 0.289, 0.278, 0.282, 0.274],
 20: [0.27, 0.279, 0.279, 0.282, 0.285],
 50: [0.271, 0.288, 0.278, 0.269, 0.266],
 100: [0.256, 0.27, 0.263, 0.256, 0.263]}
```



In [13]:

```
# 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: 4

YourExplanation:

1. Since kNN chooses based on neighbors, the decision boundary can be nonlinear.
2. There is no notion of training error for a kNN classifier since the training step simply entails remembering the training data
3. The test error could be higher or lower with a 1-NN depending on the nature of the test data.
4. The kNN algorithm is O(1) for training and O(N) for prediction, indicating that the runtime grows as more training examples are added.

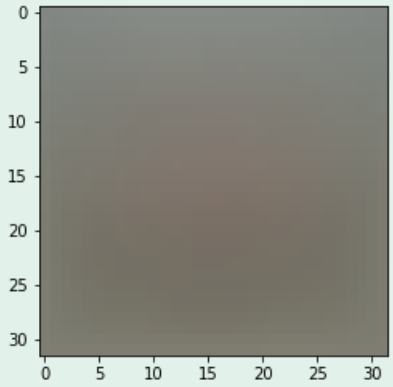
Multiclass Support Vector Machine exercise

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: 8.750410
```

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: -9.739755 analytic: -9.739755, relative error: 3.198526e-11
numerical: 9.224217 analytic: 9.224217, relative error: 7.674416e-12
numerical: 10.684165 analytic: 10.684165, relative error: 1.838012e-12
numerical: 0.729882 analytic: 0.729882, relative error: 2.383801e-10
numerical: -1.608153 analytic: -1.608153, relative error: 1.017521e-10
numerical: -43.175575 analytic: -43.175575, relative error: 5.490896e-12
numerical: 8.270273 analytic: 8.270273, relative error: 7.768179e-11
numerical: -6.250561 analytic: -6.250561, relative error: 6.226566e-12
numerical: 24.650286 analytic: 24.650286, relative error: 5.279862e-12
numerical: -6.550645 analytic: -6.550645, relative error: 2.073184e-12
numerical: 10.683086 analytic: 10.674013, relative error: 4.248284e-04
numerical: -18.659911 analytic: -18.664282, relative error: 1.171165e-04
numerical: -35.090264 analytic: -35.094027, relative error: 5.361078e-05
numerical: -7.214299 analytic: -7.215096, relative error: 5.519482e-05
numerical: -8.108428 analytic: -8.106746, relative error: 1.037801e-04
numerical: 16.658483 analytic: 16.653571, relative error: 1.474395e-04
numerical: 3.835324 analytic: 3.829262, relative error: 7.909828e-04
numerical: 19.447288 analytic: 19.435156, relative error: 3.120190e-04
numerical: 20.597232 analytic: 20.589999, relative error: 1.756298e-04
numerical: 3.934575 analytic: 3.932981, relative error: 2.025143e-04
```

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: Inspecting the svm_loss functions in linear_svm.py reveals that the hinge loss is used as the cost function for the classifier. This cost function is not differentiable about the point where the margin is zero, and as such will generate the discrepancies described above at this point.

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: 8.750410e+00 computed in 0.111653s
Vectorized loss: 8.750410e+00 computed in 0.005901s
difference: 0.000000
```

In [10]:

```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.
```

```
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.101211s
Vectorized loss and gradient: computed in 0.002974s
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.

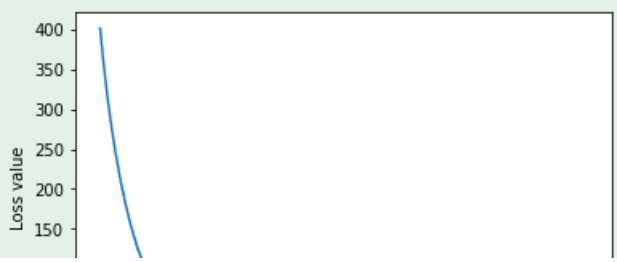
In [11]:

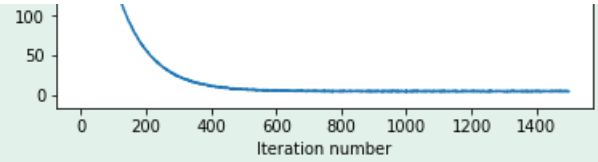
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                  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

```
iteration 0 / 1500: loss 401.294104
iteration 100 / 1500: loss 147.403520
iteration 200 / 1500: loss 56.974688
iteration 300 / 1500: loss 23.281273
iteration 400 / 1500: loss 11.500365
iteration 500 / 1500: loss 7.706428
iteration 600 / 1500: loss 5.815746
iteration 700 / 1500: loss 5.494574
iteration 800 / 1500: loss 5.191696
iteration 900 / 1500: loss 5.576933
iteration 1000 / 1500: loss 5.437777
iteration 1100 / 1500: loss 4.975029
iteration 1200 / 1500: loss 5.019156
iteration 1300 / 1500: loss 4.586635
iteration 1400 / 1500: loss 5.249953
That took 5.590317s
```

In [12]:

```
# 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 [15]:

```
# 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), ))
```

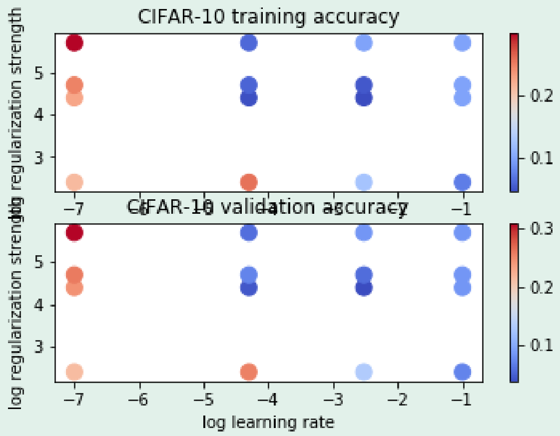
```
training accuracy: 0.209449
validation accuracy: 0.215000
```

In [25]:

```
# 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, 3e-3, 1e-1]
regularization_strengths = [2.5e2, 2.5e4, 5e4, 5e5]

# 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)*****
# Iterate through all combinations
for lr in learning_rates:
    for reg in regularization_strengths:
        print("Testing for learning rate %d and regularization %d" % (lr, reg))
        # Train the SVM
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate = lr, reg = reg, num_iters = 100,
verbose = False)
        # Predict on training and validation sets
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(lr, reg)] = (y_train_acc, y_val_acc)

        # If the current val acc is the best, update best_val and store the SVM used
        if y_val_acc > best_val:
            best_val = y_val_acc
            best_svm = svm

pass
```

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

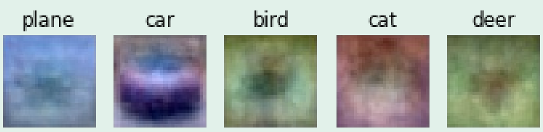
```
Testing for learning rate 0 and regularization 250
Testing for learning rate 0 and regularization 25000
Testing for learning rate 0 and regularization 50000
Testing for learning rate 0 and regularization 500000
Testing for learning rate 0 and regularization 250
Testing for learning rate 0 and regularization 25000
Testing for learning rate 0 and regularization 50000
Testing for learning rate 0 and regularization 500000
Testing for learning rate 0 and regularization 250
Testing for learning rate 0 and regularization 25000
Testing for learning rate 0 and regularization 50000
Testing for learning rate 0 and regularization 500000
Testing for learning rate 0 and regularization 250
Testing for learning rate 0 and regularization 25000
Testing for learning rate 0 and regularization 50000
Testing for learning rate 0 and regularization 500000
lr 1.000000e-07 reg 2.500000e+02 train accuracy: 0.210694 val accuracy: 0.215000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.226184 val accuracy: 0.245000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.247694 val accuracy: 0.259000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.296612 val accuracy: 0.308000
lr 5.000000e-05 reg 2.500000e+02 train accuracy: 0.255959 val accuracy: 0.256000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.049000 val accuracy: 0.046000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.061122 val accuracy: 0.074000
lr 5.000000e-05 reg 5.000000e+05 train accuracy: 0.063000 val accuracy: 0.061000
lr 3.000000e-03 reg 2.500000e+02 train accuracy: 0.127143 val accuracy: 0.134000
lr 3.000000e-03 reg 2.500000e+04 train accuracy: 0.046551 val accuracy: 0.039000
lr 3.000000e-03 reg 5.000000e+04 train accuracy: 0.054143 val accuracy: 0.059000
lr 3.000000e-03 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 2.500000e+02 train accuracy: 0.075061 val accuracy: 0.074000
lr 1.000000e-01 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-01 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.308000
```

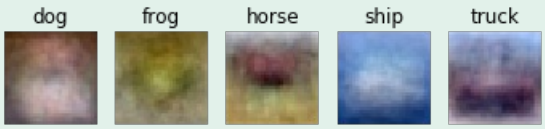


In [27]:

```python
# 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: 0.315000
```





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 weights should represent an "average" representation of all of the different images taht were used to train the classifier (and weights). For instance, the weights for "frog" should generally resemble an average configuration of all of the training frog images that were used.

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 [3]:

```
# 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: 2.313117
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer: We have chosen a uniformly random distribution of weights to initialize so the probability across classes is also uniform as p_c = 1/C. Here, C=10 so p_c = 0.1. Finally the cross-entropy loss function takes the negative log of this value s.t. we expect a loss of -log(0.1)

In [16]:

```
# 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: 1.119998 analytic: 3.785787, relative error: 5.433970e-01
numerical: -0.559127 analytic: -4.500386, relative error: 7.789800e-01
```

```
numerical: -1.046154 analytic: -0.419367, relative error: 4.276894e-01
numerical: -0.959527 analytic: -0.943623, relative error: 8.356519e-03
numerical: 0.052856 analytic: -4.993930, relative error: 1.000000e+00
numerical: 1.894545 analytic: 1.602433, relative error: 8.353256e-02
numerical: -2.168490 analytic: 0.664306, relative error: 1.000000e+00
numerical: -0.758829 analytic: -2.037473, relative error: 4.572625e-01
numerical: 0.562072 analytic: -0.404666, relative error: 1.000000e+00
numerical: -4.297420 analytic: -1.418291, relative error: 5.037219e-01
numerical: 0.687402 analytic: 1.841398, relative error: 4.563414e-01
numerical: 0.403487 analytic: -1.453005, relative error: 1.000000e+00
numerical: 1.115321 analytic: -1.111024, relative error: 1.000000e+00
numerical: -3.150023 analytic: -2.234717, relative error: 1.699815e-01
numerical: 0.002194 analytic: 1.178666, relative error: 9.962846e-01
numerical: 1.025466 analytic: -0.359415, relative error: 1.000000e+00
numerical: -0.348525 analytic: 2.480224, relative error: 1.000000e+00
numerical: 2.077976 analytic: 1.088842, relative error: 3.123431e-01
numerical: 0.039545 analytic: -0.654690, relative error: 1.000000e+00
numerical: 0.738425 analytic: -0.103696, relative error: 1.000000e+00
```

In [19]:

```
# 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 Fr}obenius 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: 2.313117e+00 computed in 0.018949s
vectorized loss: 2.313117e+00 computed in 0.001996s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [23]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of 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 learning_rates:
    for reg in regularization_strengths:
        classifier = Softmax()
        classifier.train(X_train, y_train, learning_rate = lr, reg = reg, num_iters = 1000, verbose
= False)
        y_train_pred = classifier.predict(X_train)
        y_val_pred = classifier.predict(X_val)
        train_acc = np.mean(y_train == y_train_pred)
        val_acc = np.mean(y_val == y_val_pred)
        results[(lr, reg)] = (train_acc, val_acc)
```

```
        if val_acc > best_val:
            best_val = val_acc
            best_softmax = classifier
pass

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

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

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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.337571 val accuracy: 0.341000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.324837 val accuracy: 0.338000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.350020 val accuracy: 0.367000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.314898 val accuracy: 0.339000
best validation accuracy achieved during cross-validation: 0.367000
```

In [24]:

```
# 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: 0.351000
```

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: False

YourExplanation: The training loss for an SVM uses the hinge loss function, wihle the softmax classifier uses cross-entropy. Because the hinge loss is zero for any datapoint which clears the margin by more than 1 will produce zero incremental loss. Meanwhile, the cross-entropy loss will account for the influence of all datapoints in its loss calculation.

In [25]:

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

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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

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 [4]:

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

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

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

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

Difference between your scores and correct scores:
3.6802720496109664e-08
```

Forward pass: compute loss

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

In [8]:

```
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 [14]:

```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

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

Train the network

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

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

In [17]:

```
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 [20]:

```
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 2.302970
iteration 100 / 1000: loss 2.302474
iteration 200 / 1000: loss 2.297076
iteration 300 / 1000: loss 2.257328
iteration 400 / 1000: loss 2.230484
iteration 500 / 1000: loss 2.150620
iteration 600 / 1000: loss 2.080736
iteration 700 / 1000: loss 2.054914
iteration 800 / 1000: loss 1.979290
iteration 900 / 1000: loss 2.039101
Validation accuracy:  0.287
```

Debug the training

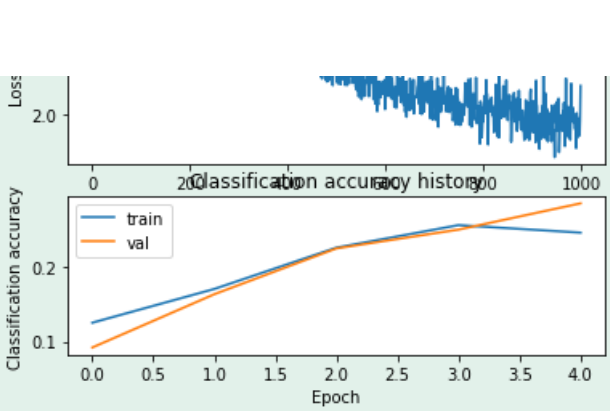
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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 [21]:

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

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



In [8]:

```
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: Specified different values for the learning rate, regularization coefficient, and the size of the hidden layer used in the netwrok.

In [6]:

```
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)*****
input_dim = X_train.shape[1]
```

```
batch_size = 200

C = 10

# Initialize variables to store best results
best_val_acc = -1
best_lr = -1
best_reg = -1
best_hs = -1

# Parameters
hidden_layer_sizes = [50, 100, 500]
learning_rates = [0.01, 0.1, 0.5]
regularization_strengths = [0.01, 0.1, 1.0]

for lr in learning_rates:
    for reg in regularization_strengths:
        for hs in hidden_layer_sizes:
            print("Testing lr={:.2f}, reg={:.2f}, hs={}".format(lr,reg,hs))
            nn = TwoLayerNet(input_size=input_dim, hidden_size=hs, output_size=C)
            nn.train(X_train, y_train, X_val, y_val, batch_size = batch_size,
                    num_iters = 1500, learning_rate = lr, reg = reg, learning_rate_decay = 0.95, ve
bose = False)

            train_results = nn.predict(X_train) == y_train
            train_acc = train_results.mean()
            val_results = nn.predict(X_val) == y_val
            val_acc = val_results.mean()

            if val_acc > best_val_acc:
                best_val_acc = val_acc
                best_lr = lr
                best_reg = reg
                best_hs = hs
                best_net = nn

print(best_val_acc, best_lr, best_reg, best_hs)

pass

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

```
Testing lr=0.01, reg=0.01, hs=50
Testing lr=0.01, reg=0.01, hs=100
Testing lr=0.01, reg=0.01, hs=500
Testing lr=0.01, reg=0.10, hs=50
Testing lr=0.01, reg=0.10, hs=100
Testing lr=0.01, reg=0.10, hs=500
Testing lr=0.01, reg=1.00, hs=50
Testing lr=0.01, reg=1.00, hs=100
Testing lr=0.01, reg=1.00, hs=500
Testing lr=0.10, reg=0.01, hs=50
Testing lr=0.10, reg=0.01, hs=100
Testing lr=0.10, reg=0.01, hs=500
```

```
C:\Users\Anand Natu\Desktop\CS231n-Assignments\assignment1\cs231n\classifiers\neural_net.py:111: R
untimeWarning: overflow encountered in multiply
  loss = loss + reg*(np.sum(W1*W1) + np.sum(W2*W2))
```

```
Testing lr=0.10, reg=0.10, hs=50
Testing lr=0.10, reg=0.10, hs=100
```

```
C:\Users\Anand Natu\AppData\Local\conda\conda\envs\cs231n\lib\site-
packages\numpy\core\fromnumeric.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```

```
Testing lr=0.10, reg=0.10, hs=500
Testing lr=0.10, reg=1.00, hs=50
Testing lr=0.10, reg=1.00, hs=100
Testing lr=0.10, reg=1.00, hs=500
Testing lr=0.50, reg=0.01, hs=50
```

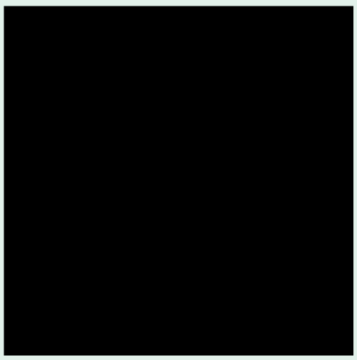
```
C:\Users\Anand Natu\Desktop\CS231n-Assignments\assignment1\cs231n\classifiers\neural_net.py:111: R
untimeWarning: overflow encountered in double_scalars
  loss = loss + reg*(np.sum(W1*W1) + np.sum(W2*W2))
```

```
Testing lr=0.50, reg=0.01, hs=100
Testing lr=0.50, reg=0.01, hs=500
```

```
Testing lr=0.50, reg=0.10, hs=50
Testing lr=0.50, reg=0.10, hs=100
Testing lr=0.50, reg=0.10, hs=500
Testing lr=0.50, reg=1.00, hs=50
Testing lr=0.50, reg=1.00, hs=100
Testing lr=0.50, reg=1.00, hs=500
0.087 0.01 0.01 50
```

In [9]:

```
# 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 [10]:

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

```
Test accuracy:  0.103
```

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: A significantly lower testing accuracy is thes result of a bias error, i.e. overfitting. Option 1 (using a larger training dataset) can help address this issue by providing more examples and thus (barring any data bias issues) adding more generality to the model's training process. Option 3 (applying regularization) dampens the update of the model weights which in turn mitigates overfitting resulting from excessively tuned weights.

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 [9]:

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

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
    for reg in regularization_strengths:
        classifier = LinearSVM()
        loss_history = classifier.train(X_train_feats, y_train, learning_rate = lr,
                                        reg = reg, num_iters = 2000, verbose = False)
        train_results = classifier.predict(X_train_feats) == y_train
        train_accuracy = train_results.mean()
        val_results = classifier.predict(X_val_feats) == y_val
        val_accuracy = val_results.mean()

        results[(lr,reg)] = (lr,reg)

        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = classifier
pass

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

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

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

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 50000.000000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.000000 val accuracy: 500000.000000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.000000 val accuracy: 5000000.000000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 50000.000000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.000000 val accuracy: 500000.000000
```

```
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.000000 val accuracy: 5000000.000000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 50000.000000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.000000 val accuracy: 500000.000000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.000000 val accuracy: 5000000.000000
best validation accuracy achieved during cross-validation: 0.418000
```

In [10]:

```
# 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)
```

```
0.413
```

In [11]:

```
# 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: Inspection of the misclassification results is consistent with our HOG feature representation of the images, which captures spatial orientation and image texture without regard for color composition. Accordingly, we see misclassified images which have spatial and edge features that resemble the predicted class, but differ significantly in color makeup.

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 [26]:

```
from cs231n.classifiers.neural_net import TwoLayerNet

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

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
learning_rates = [0.3, 0.5, 0.7, 0.9]
regularization_strengths = [0.001, 0.01, 0.05, 0.1]
best_val = -1

for lr in learning_rates:
    for reg in regularization_strengths:
        nn = TwoLayerNet(input_dim, hidden_dim, num_classes)
        nn.train(X_train_feats, y_train, X_val_feats, y_val,
                 num_iters = 3000, batch_size = 200, learning_rate = lr, reg = reg, verbose = False,
                 learning_rate_decay = 0.95)

        train_results = nn.predict(X_train_feats) == y_train
        train_accuracy = train_results.mean()
        val_results = nn.predict(X_val_feats) == y_val
        val_accuracy = val_results.mean()

        if val_accuracy > best_val:
            best_val = val_accuracy
            best_net = nn
pass

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

In [27]:

```
# 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)
```

```
0.595
```

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)*****
                # L2 distance between test example i and train example j
                dists[i,j] = np.sqrt(np.sum((X[i]-self.X_train[j])**2))
                pass

                # *****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)*****
            # Compute the partially vectorized L2 distance and assign to distance matrix
            # (ith test example against all training examples)
            sqDist = np.sum((X[i]-self.X_train)**2,axis=1)
            dists[i,:]=np.sqrt(sqDist)
            pass

            # *****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)*****
        # Broadcast sums on train and test terms
        xTrain_sq = (X**2).sum(axis = 1, keepdims = 1)
        xTest_sq = (self.X_train**2).sum(axis = 1)
        cross_term = 2*np.dot(X, self.X_train.T)

        # Sum in quadrature to produce overall distance matrix
        dists = np.sqrt(xTrain_sq + xTest_sq - cross_term)
        pass

        # *****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)*****
            # For each test example, sort that row of the distance matrix
            # according to ascending distance
            sortedDists = np.argsort(dists[i,:])
            # Find the k nearest neighbors
            kNearest = sortedDists[0:k]
            # Reference y_train to find the labels of these neighbors
            closest_y = self.y_train[kNearest]

            pass

            # *****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)*****
            # Take the de-duped set of labels along with the frequency of each label
            uniques, cts = np.unique(closest_y, return_counts = True)
            # Form the predicted label based on the most frequent label
            # (the argmax function will automatically take the first occurrence
            # to break ties which should correspond to the smaller label)
            y_pred[i] = uniques[np.argmax(cts)]
            pass

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

        return y_pred
```

2 linear_classifier.py

```
from __future__ import print_function

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


class LinearClassifier(object):

    def __init__(self):
        self.W = None

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

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

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

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

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

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

            # W = W - eta*grad(W)
            self.W = self.W - learning_rate*grad
            pass

            # *****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.dot(self.W)
        # Take the max score as the predicted class
        y_pred = np.argmax(scores, axis=1)
        pass

        # *****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
                # Calculate gradient dW for positive margin
                # Decrement weight on predicted class
                dW[:, y[i]] -= X[i,:]
                # Increment weight for jth class (true class)
                dW[:, j] += 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
    # Normalize the derivative across number of training examples
    dW /= num_train

    # Add regularization to the loss.
    loss += reg * np.sum(W * W)
    # Regularize the gradient
    dW += reg*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)*****
    # IMPLEMENTED INLINE - SEE ABOVE!
    pass

    # *****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 = np.arange(X.shape[0])
    # Compute the entire score matrix
    scores = np.dot(X, W)
    # Pull actual class labels for X
    correct_class_scores = scores[N, y]
    # Compute the margin for all scores against actual labels
    margin = np.maximum(0, 1 + (scores - correct_class_scores[:, np.newaxis]))
    # Do not count y_i in the margin
    margin[N, y] = 0
    # Take sum across training samples
    marginSum = np.sum(margin, axis = 1)
    # Average across training samples and regularize loss
    loss = np.mean(marginSum) + 0.5*reg*np.sum(W*W)
    pass

    # *****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)*****
    # Convert to zero-one loss (gradient of hinge loss) and sum across all training samples
    binaryMargins = margin
    binaryMargins[margin>0] = 1
    binaryMargins[N, y] = -np.sum(binaryMargins, axis = 1).T
    # Compute gradient, normalize, and regularize
    dW = np.dot(X.T, binaryMargins)
    dW = dW / X.shape[0]
    dW = dW + 2*reg*W
    pass

    # *****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)*****
    # Num classes = num cols of w
    C = W.shape[1]
    # Number of examples
    N = X.shape[0]

    dscores = np.zeros((N,C))

    for i in range(N):
        # Calculate the scores and subtract the max (predicted class)
        scores = np.dot(X[i], W)
        scores = scores - np.max(scores)
        # Calculate probability w/ numerical stability trick
        prob = np.exp(scores)/np.sum(np.exp(scores))

        binary_y = np.zeros(C)
        # Subtract the probabilities from the true example class for gradient
        binary_y[y[i]] = 1
        dscores[i] = prob - binary_y

        # Calculate loss for the example, add to total
        L_i = -np.log(prob[y[i]])
        loss = loss + L_i

    # Normalize and regularize loss
    loss = loss / N
    loss = loss + (0.5*reg*np.sum(W*W))

    # Calculate gradient, normalize / regularize
    dW = np.dot(X.T, dscores)/N
    dW = dW + (reg*W)

    pass

    # *****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)*****
    N = X.shape[0]
    C = W.shape[1]

    # Calculate scores and probability (increase dimension of max for broadcasting)
    scores = np.dot(X, W)
    scores = scores - np.max(scores, axis=1)[:, np.newaxis]
    prob = np.exp(scores)/np.sum(np.exp(scores), axis=1)[:, np.newaxis]

    # Calculate and regularize loss
    loss = -np.log(prob[np.arange(N), y])
    loss = np.mean(loss)
    loss = loss + (0.5*reg*np.sum(W*W))

    # Calculate score differences for gradient
    dscores = prob
    dscores[np.arange(N), y] = dscores[np.arange(N), y] - 1
    dscores = dscores/N

    dW = np.dot(X.T, dscores)
    dW = dW + (reg*W)


    pass

    # *****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)*****
        # Compute activated forward step for first hidden layer
        # (use np.maximum for elementwise max)
        h1 = np.maximum(0, np.dot(X, W1) + b1)
        # Compute final scores at second (output) layer
        scores = np.dot(h1, W2) + b2
        pass

        # *****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)*****
        # Subtract predicted class from scores and calculate probabilities
        scores = scores - np.max(scores, axis = 1)[:, np.newaxis]
        probs = np.exp(scores)/np.sum(np.exp(scores),axis=1)[:, np.newaxis]

        # Compute loss and add regularization
        loss = -np.log(probs[np.arange(N), y])
        loss = np.mean(loss)
        loss = loss + reg*(np.sum(W1*W1) + np.sum(W2*W2))
        pass

        # *****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)*****
        # Compute delta for the gradient (same as from softmax)
        dscores = probs
        dscores[np.arange(N), y] = dscores[np.arange(N), y] - 1
        dscores = dscores / N

        # Calculate gradient at output layer
        dW2 = np.dot(h1.T, dscores)
        dW2 = dW2 + 2*reg*W2
        db2 = np.sum(dscores, axis=0)
        # Propagate into hidden layer
        dh = np.dot(dscores, W2.T)
        # Apply mask
        dh[h1<=0] = 0
        # Gradient at input layer
        dW1 = np.dot(X.T, dh)
        dW1 = dW1 + 2*reg*W1
        db1 = np.sum(dh, axis = 0)

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

        pass

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

```
        return loss, grads

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

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

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

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

            #########################################################################
            # TODO: Create a random minibatch of training data and labels, storing  #
            # them in X_batch and y_batch respectively.                             #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            batch_idx = np.random.choice(num_train, batch_size)
            X_batch = X[batch_idx]
            y_batch = y[batch_idx]
            pass

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

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

            #########################################################################
            # TODO: Use the gradients in the grads dictionary to update the         #
            # parameters of the network (stored in the dictionary self.params)      #
            # using stochastic gradient descent. You'll need to use the gradients   #
            # stored in the grads dictionary defined above.                         #
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            self.params["W1"] = self.params["W1"] - learning_rate*grads["W1"]
            self.params["W2"] = self.params["W2"] - learning_rate*grads["W2"]
            self.params["b1"] = self.params["b1"] - learning_rate*grads["b1"]
            self.params["b2"] = self.params["b2"] - learning_rate*grads["b2"]
            pass

            # *****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)*****
        y_pred = np.argmax(self.loss(X), axis = 1)
        pass

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

        return y_pred
```