svm

October 27, 2019

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

```
In [1]: # Run some setup code for this notebook.
        import random
        import numpy as np
        from cs231n.data_utils import load_CIFAR10
        import matplotlib.pyplot as plt

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

```
In [3]: # Visualize some examples from the dataset.
        # We show a few examples of training images from each class.
        classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck
        num_classes = len(classes)
        samples_per_class = 7
        for y, cls in enumerate(classes):
            idxs = np.flatnonzero(y_train == y)
            idxs = np.random.choice(idxs, samples_per_class, replace=False)
            for i, idx in enumerate(idxs):
                plt_idx = i * num_classes + y + 1
                plt.subplot(samples_per_class, num_classes, plt_idx)
                plt.imshow(X_train[idx].astype('uint8'))
                plt.axis('off')
                if i == 0:
                    plt.title(cls)
        plt.show()
```

```
In [4]: # Split the data into train, val, and test sets. In addition we will
        # create a small development set as a subset of the training data;
        # we can use this for development so our code runs faster.
        num_training = 49000
        num_validation = 1000
        num_test = 1000
        num_dev = 500

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

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

        # We will also make a development set, which is a small subset of
        # the training set.
        mask = np.random.choice(num_training, num_dev, replace=False)
        X_dev = X_train[mask]
        y_dev = y_train[mask]
        # We use the first num_test points of the original test set as our
        # test set.
        mask = range(num_test)
        X_test = X_test[mask]
        y_test = y_test[mask]

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

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

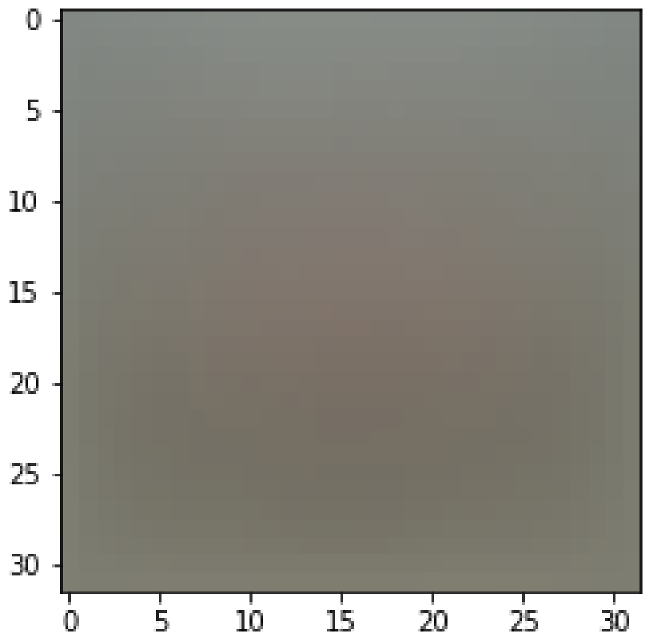
```
In [5]: # Preprocessing: reshape the image data into rows
        X_train = np.reshape(X_train, (X_train.shape[0], -1))
        X_val = np.reshape(X_val, (X_val.shape[0], -1))
        X_test = np.reshape(X_test, (X_test.shape[0], -1))
        X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

        # As a sanity check, print out the shapes of the data
        print('Training data shape: ', X_train.shape)
        print('Validation data shape: ', X_val.shape)
        print('Test data shape: ', X_test.shape)
        print('dev data shape: ', X_dev.shape)
```

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

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

        # second: subtract the mean image from train and test data
```

```
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

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

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

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



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

1.2 SVM Classifier

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

As you can see, we have prefilled the function `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: 9.336636
```

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

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

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

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

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

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

```
numerical: 7.876000 analytic: 7.876000, relative error: 3.425278e-11
numerical: -12.832158 analytic: -12.832158, relative error: 4.464038e-11
numerical: 21.414694 analytic: 21.414694, relative error: 1.609486e-12
numerical: 6.618000 analytic: 6.618000, relative error: 7.277272e-11
numerical: -9.108000 analytic: -9.108000, relative error: 3.774173e-12
numerical: -5.164987 analytic: -5.164987, relative error: 1.390239e-11
numerical: 11.469543 analytic: 11.469543, relative error: 5.951884e-13
numerical: 24.513100 analytic: 24.513100, relative error: 1.241126e-11
numerical: -5.814885 analytic: -5.814885, relative error: 2.567474e-11
numerical: 17.274196 analytic: 17.274196, relative error: 1.874090e-11
```

```
numerical: 10.964158 analytic: 10.949591, relative error: 6.647637e-04
numerical: 23.681462 analytic: 23.681462, relative error: 8.022366e-12
numerical: 11.622054 analytic: 11.620252, relative error: 7.753080e-05
numerical: 10.976579 analytic: 10.976579, relative error: 3.293385e-11
numerical: -9.894968 analytic: -9.894968, relative error: 1.565344e-11
numerical: -19.034578 analytic: -19.034578, relative error: 1.556967e-11
numerical: -5.490900 analytic: -5.490900, relative error: 5.151739e-11
numerical: 36.600008 analytic: 36.600008, relative error: 1.788587e-12
numerical: -63.266903 analytic: -63.266903, relative error: 7.659570e-12
numerical: 1.063658 analytic: 1.063658, relative error: 4.888557e-10
```

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 : The discrepancy with gradient checking occurs when the margin is strictly equal to zero or is close to zero. For two-class problem, this is the case when the difference between scores for the first and second classes is equal to one. This is actually not a big deal, the weights can still be updated in such case.

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

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

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

```
Naive loss: 9.336636e+00 computed in 0.145501s
Vectorized loss: 9.336636e+00 computed in 0.004960s
difference: 0.000000
```

```
In [10]: # Complete the implementation of svm_loss_vectorized, and compute the gradient
         # of the loss function in a vectorized way.

         # The naive implementation and the vectorized implementation should match, but
         # the vectorized version should still be much faster.
         tic = time.time()
         _, 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 0.136826s
Vectorized loss and gradient: computed in 0.005207s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

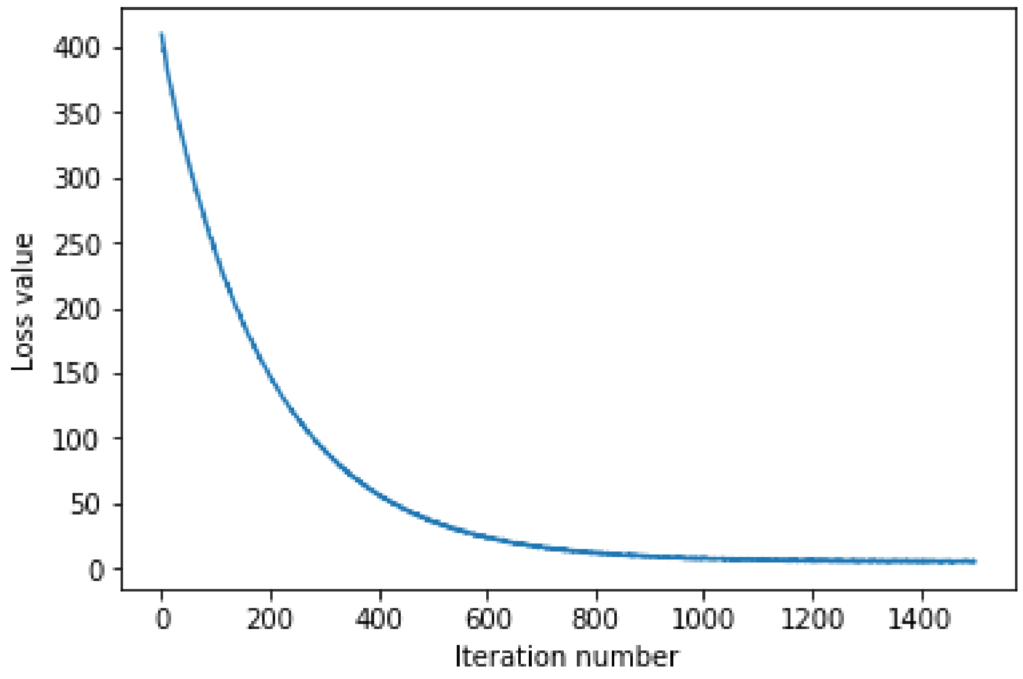
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

```
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 409.782555
iteration 100 / 1500: loss 242.075974
iteration 200 / 1500: loss 147.242236
iteration 300 / 1500: loss 90.965109
iteration 400 / 1500: loss 56.819261
iteration 500 / 1500: loss 35.717123
iteration 600 / 1500: loss 23.825037
iteration 700 / 1500: loss 16.687087
iteration 800 / 1500: loss 11.569955
iteration 900 / 1500: loss 9.042283
iteration 1000 / 1500: loss 7.337243
iteration 1100 / 1500: loss 6.580811
iteration 1200 / 1500: loss 5.764976
iteration 1300 / 1500: loss 5.736847
iteration 1400 / 1500: loss 5.801470
```

```
That took 5.297017s
```

```
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 [13]: # 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: 0.383714
validation accuracy: 0.391000
```

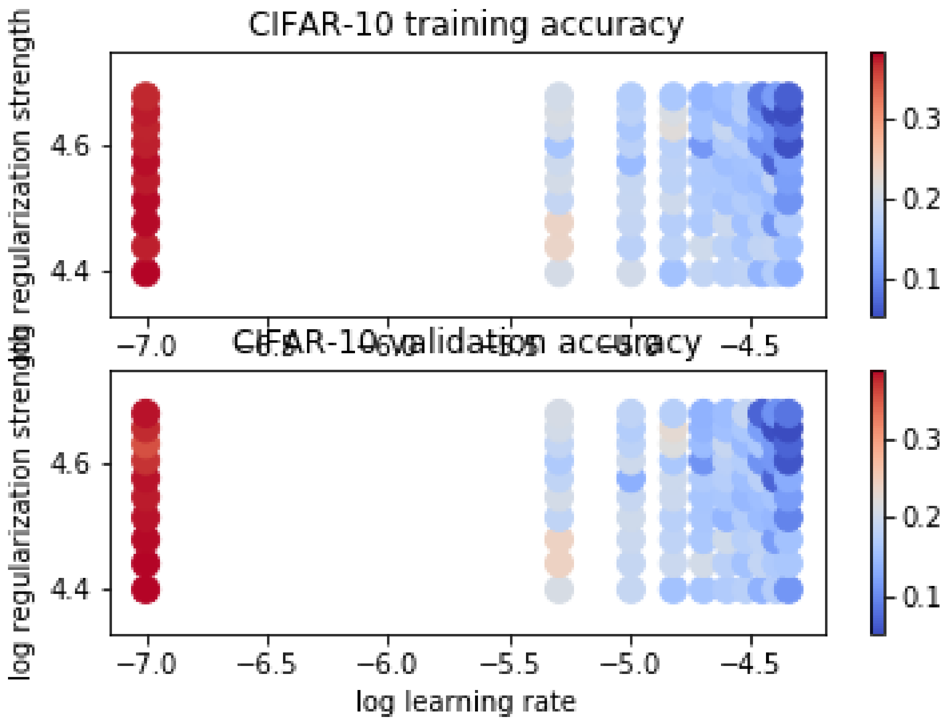
```
In [14]: # Use the validation set to tune hyperparameters (regularization strength and
         # learning rate). You should experiment with different ranges for the learning
         # rates and regularization strengths; if you are careful you should be able to
         # get a classification accuracy of about 0.39 on the validation set.
```

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

learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

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

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

num_splt = 10
for i in range(num_splt):
    for j in range(num_splt):
        lr = (learning_rates[1] - learning_rates[0]) * i / num_splt + learning_rates[0]
        reg = (regularization_strengths[1] - regularization_strengths[0]) * j / num_spl
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=reg,num_iters=150

        y_train_pred = svm.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        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_svm = svm

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

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

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

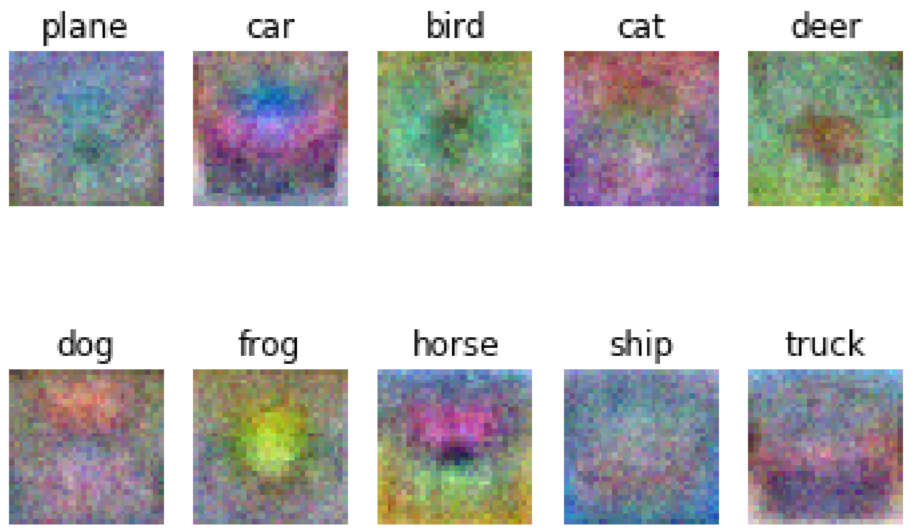
```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381286 val accuracy: 0.389000
lr 1.000000e-07 reg 2.750000e+04 train accuracy: 0.373469 val accuracy: 0.390000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.378776 val accuracy: 0.388000
lr 1.000000e-07 reg 3.250000e+04 train accuracy: 0.378980 val accuracy: 0.385000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.373878 val accuracy: 0.383000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.377633 val accuracy: 0.386000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.372735 val accuracy: 0.374000
lr 1.000000e-07 reg 4.250000e+04 train accuracy: 0.371796 val accuracy: 0.357000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.374327 val accuracy: 0.377000
lr 1.000000e-07 reg 4.750000e+04 train accuracy: 0.370898 val accuracy: 0.385000
lr 5.090000e-06 reg 2.500000e+04 train accuracy: 0.206041 val accuracy: 0.212000
lr 5.090000e-06 reg 2.750000e+04 train accuracy: 0.235755 val accuracy: 0.244000
lr 5.090000e-06 reg 3.000000e+04 train accuracy: 0.237388 val accuracy: 0.244000
lr 5.090000e-06 reg 3.250000e+04 train accuracy: 0.189143 val accuracy: 0.187000
lr 5.090000e-06 reg 3.500000e+04 train accuracy: 0.205245 val accuracy: 0.208000
lr 5.090000e-06 reg 3.750000e+04 train accuracy: 0.197490 val accuracy: 0.187000
lr 5.090000e-06 reg 4.000000e+04 train accuracy: 0.161041 val accuracy: 0.170000
lr 5.090000e-06 reg 4.250000e+04 train accuracy: 0.202694 val accuracy: 0.190000
lr 5.090000e-06 reg 4.500000e+04 train accuracy: 0.210816 val accuracy: 0.206000
lr 5.090000e-06 reg 4.750000e+04 train accuracy: 0.204143 val accuracy: 0.211000
lr 1.008000e-05 reg 2.500000e+04 train accuracy: 0.203224 val accuracy: 0.192000
lr 1.008000e-05 reg 2.750000e+04 train accuracy: 0.178735 val accuracy: 0.192000
lr 1.008000e-05 reg 3.000000e+04 train accuracy: 0.188510 val accuracy: 0.197000
lr 1.008000e-05 reg 3.250000e+04 train accuracy: 0.190939 val accuracy: 0.202000
lr 1.008000e-05 reg 3.500000e+04 train accuracy: 0.189388 val accuracy: 0.194000
lr 1.008000e-05 reg 3.750000e+04 train accuracy: 0.147531 val accuracy: 0.137000
lr 1.008000e-05 reg 4.000000e+04 train accuracy: 0.178306 val accuracy: 0.201000
lr 1.008000e-05 reg 4.250000e+04 train accuracy: 0.163388 val accuracy: 0.183000
lr 1.008000e-05 reg 4.500000e+04 train accuracy: 0.179286 val accuracy: 0.173000
lr 1.008000e-05 reg 4.750000e+04 train accuracy: 0.172265 val accuracy: 0.185000
lr 1.507000e-05 reg 2.500000e+04 train accuracy: 0.156204 val accuracy: 0.158000
lr 1.507000e-05 reg 2.750000e+04 train accuracy: 0.180959 val accuracy: 0.197000
lr 1.507000e-05 reg 3.000000e+04 train accuracy: 0.174388 val accuracy: 0.186000
lr 1.507000e-05 reg 3.250000e+04 train accuracy: 0.198347 val accuracy: 0.179000
lr 1.507000e-05 reg 3.500000e+04 train accuracy: 0.181347 val accuracy: 0.198000
lr 1.507000e-05 reg 3.750000e+04 train accuracy: 0.182857 val accuracy: 0.200000
lr 1.507000e-05 reg 4.000000e+04 train accuracy: 0.177429 val accuracy: 0.166000
lr 1.507000e-05 reg 4.250000e+04 train accuracy: 0.220163 val accuracy: 0.220000
lr 1.507000e-05 reg 4.500000e+04 train accuracy: 0.210939 val accuracy: 0.231000
lr 1.507000e-05 reg 4.750000e+04 train accuracy: 0.164735 val accuracy: 0.177000
lr 2.006000e-05 reg 2.500000e+04 train accuracy: 0.185776 val accuracy: 0.161000
lr 2.006000e-05 reg 2.750000e+04 train accuracy: 0.199592 val accuracy: 0.206000
lr 2.006000e-05 reg 3.000000e+04 train accuracy: 0.166286 val accuracy: 0.165000
lr 2.006000e-05 reg 3.250000e+04 train accuracy: 0.170898 val accuracy: 0.163000
lr 2.006000e-05 reg 3.500000e+04 train accuracy: 0.165531 val accuracy: 0.162000
lr 2.006000e-05 reg 3.750000e+04 train accuracy: 0.167245 val accuracy: 0.170000
lr 2.006000e-05 reg 4.000000e+04 train accuracy: 0.112694 val accuracy: 0.109000
lr 2.006000e-05 reg 4.250000e+04 train accuracy: 0.156776 val accuracy: 0.149000
lr 2.006000e-05 reg 4.500000e+04 train accuracy: 0.156184 val accuracy: 0.141000
lr 2.006000e-05 reg 4.750000e+04 train accuracy: 0.142367 val accuracy: 0.139000
lr 2.505000e-05 reg 2.500000e+04 train accuracy: 0.179694 val accuracy: 0.171000
lr 2.505000e-05 reg 2.750000e+04 train accuracy: 0.180469 val accuracy: 0.180000
lr 2.505000e-05 reg 3.000000e+04 train accuracy: 0.194939 val accuracy: 0.203000
lr 2.505000e-05 reg 3.250000e+04 train accuracy: 0.158327 val accuracy: 0.143000
lr 2.505000e-05 reg 3.500000e+04 train accuracy: 0.164673 val accuracy: 0.164000
lr 2.505000e-05 reg 3.750000e+04 train accuracy: 0.151306 val accuracy: 0.176000
lr 2.505000e-05 reg 4.000000e+04 train accuracy: 0.167367 val accuracy: 0.180000
lr 2.505000e-05 reg 4.250000e+04 train accuracy: 0.189612 val accuracy: 0.193000
lr 2.505000e-05 reg 4.500000e+04 train accuracy: 0.144918 val accuracy: 0.155000
lr 2.505000e-05 reg 4.750000e+04 train accuracy: 0.152224 val accuracy: 0.151000
lr 3.004000e-05 reg 2.500000e+04 train accuracy: 0.184163 val accuracy: 0.179000
lr 3.004000e-05 reg 2.750000e+04 train accuracy: 0.163245 val accuracy: 0.157000
lr 3.004000e-05 reg 3.000000e+04 train accuracy: 0.155306 val accuracy: 0.177000
lr 3.004000e-05 reg 3.250000e+04 train accuracy: 0.170408 val accuracy: 0.172000
lr 3.004000e-05 reg 3.500000e+04 train accuracy: 0.153531 val accuracy: 0.146000
lr 3.004000e-05 reg 3.750000e+04 train accuracy: 0.156245 val accuracy: 0.154000
lr 3.004000e-05 reg 4.000000e+04 train accuracy: 0.174490 val accuracy: 0.174000
lr 3.004000e-05 reg 4.250000e+04 train accuracy: 0.150102 val accuracy: 0.147000
lr 3.004000e-05 reg 4.500000e+04 train accuracy: 0.174898 val accuracy: 0.165000
lr 3.004000e-05 reg 4.750000e+04 train accuracy: 0.171694 val accuracy: 0.191000
lr 3.503000e-05 reg 2.500000e+04 train accuracy: 0.141000 val accuracy: 0.139000
lr 3.503000e-05 reg 2.750000e+04 train accuracy: 0.186306 val accuracy: 0.190000
lr 3.503000e-05 reg 3.000000e+04 train accuracy: 0.175980 val accuracy: 0.186000
lr 3.503000e-05 reg 3.250000e+04 train accuracy: 0.162551 val accuracy: 0.150000
lr 3.503000e-05 reg 3.500000e+04 train accuracy: 0.148571 val accuracy: 0.153000
lr 3.503000e-05 reg 3.750000e+04 train accuracy: 0.160612 val accuracy: 0.180000
lr 3.503000e-05 reg 4.000000e+04 train accuracy: 0.106673 val accuracy: 0.108000
lr 3.503000e-05 reg 4.250000e+04 train accuracy: 0.134776 val accuracy: 0.165000
lr 3.503000e-05 reg 4.500000e+04 train accuracy: 0.109367 val accuracy: 0.119000
lr 3.503000e-05 reg 4.750000e+04 train accuracy: 0.088265 val accuracy: 0.070000
lr 4.002000e-05 reg 2.500000e+04 train accuracy: 0.171898 val accuracy: 0.175000
lr 4.002000e-05 reg 2.750000e+04 train accuracy: 0.188980 val accuracy: 0.170000
lr 4.002000e-05 reg 3.000000e+04 train accuracy: 0.110653 val accuracy: 0.118000
lr 4.002000e-05 reg 3.250000e+04 train accuracy: 0.147224 val accuracy: 0.144000
lr 4.002000e-05 reg 3.500000e+04 train accuracy: 0.181796 val accuracy: 0.161000
lr 4.002000e-05 reg 3.750000e+04 train accuracy: 0.072245 val accuracy: 0.070000
lr 4.002000e-05 reg 4.000000e+04 train accuracy: 0.109388 val accuracy: 0.113000
lr 4.002000e-05 reg 4.250000e+04 train accuracy: 0.110918 val accuracy: 0.114000
lr 4.002000e-05 reg 4.500000e+04 train accuracy: 0.056490 val accuracy: 0.053000
lr 4.002000e-05 reg 4.750000e+04 train accuracy: 0.118163 val accuracy: 0.107000
lr 4.501000e-05 reg 2.500000e+04 train accuracy: 0.135041 val accuracy: 0.111000
lr 4.501000e-05 reg 2.750000e+04 train accuracy: 0.153122 val accuracy: 0.164000
lr 4.501000e-05 reg 3.000000e+04 train accuracy: 0.169898 val accuracy: 0.157000
lr 4.501000e-05 reg 3.250000e+04 train accuracy: 0.108939 val accuracy: 0.096000
lr 4.501000e-05 reg 3.500000e+04 train accuracy: 0.119531 val accuracy: 0.117000
lr 4.501000e-05 reg 3.750000e+04 train accuracy: 0.124857 val accuracy: 0.132000
lr 4.501000e-05 reg 4.000000e+04 train accuracy: 0.058714 val accuracy: 0.058000
lr 4.501000e-05 reg 4.250000e+04 train accuracy: 0.080163 val accuracy: 0.070000
lr 4.501000e-05 reg 4.500000e+04 train accuracy: 0.053673 val accuracy: 0.051000
lr 4.501000e-05 reg 4.750000e+04 train accuracy: 0.068816 val accuracy: 0.083000
best validation accuracy achieved during cross-validation: 0.390000
```

```
In [15]: # Visualize the cross-validation results
         import math
         x_scatter = [math.log10(x[0]) for x in results]
         y_scatter = [math.log10(x[1]) for x in results]

         # plot training accuracy
         marker_size = 100
         colors = [results[x][0] for x in results]
         plt.subplot(2, 1, 1)
         plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
         plt.colorbar()
         plt.xlabel('log learning rate')
         plt.ylabel('log regularization strength')
         plt.title('CIFAR-10 training accuracy')

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



```
In [16]: # 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: 0.360000
```

```
In [17]: # Visualize the learned weights for each class.
         # Depending on your choice of learning rate and regularization strength, these may
         # or may not be nice to look at.
         w = best_svm.W[:-1,:] # strip out the bias
         w = w.reshape(32, 32, 3, 10)
         w_min, w_max = np.min(w), np.max(w)
         classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truc
         for i in range(10):
             plt.subplot(2, 5, i + 1)

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



Inline question 2

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

Your Answer : *The weights serve as a sort of templates for each class and the weight matrices convey some general features of the representatives of each class through visualizing.*