svm

June 7, 2020

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 [5]: # 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
```

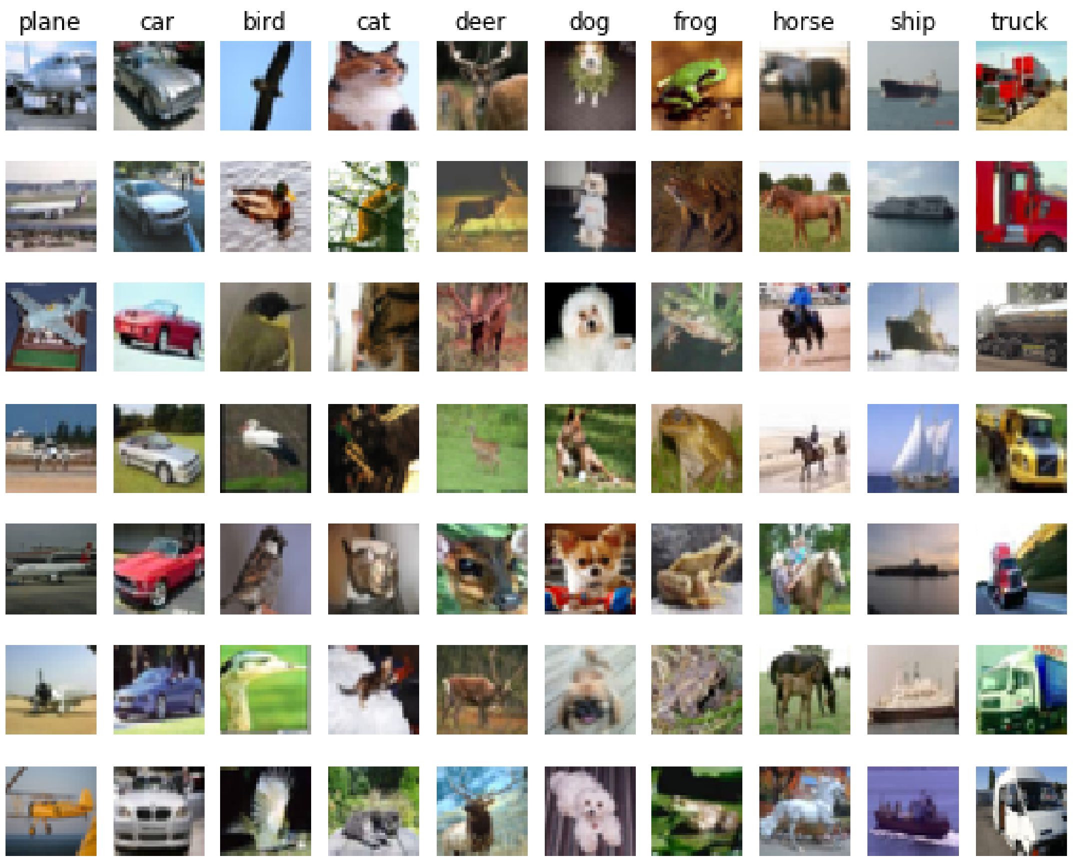
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 CIFAR-10 Data Loading and Preprocessing

```
In [6]: # Load the raw CIFAR-10 data.
        cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

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



```
In [8]: # 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,)
```

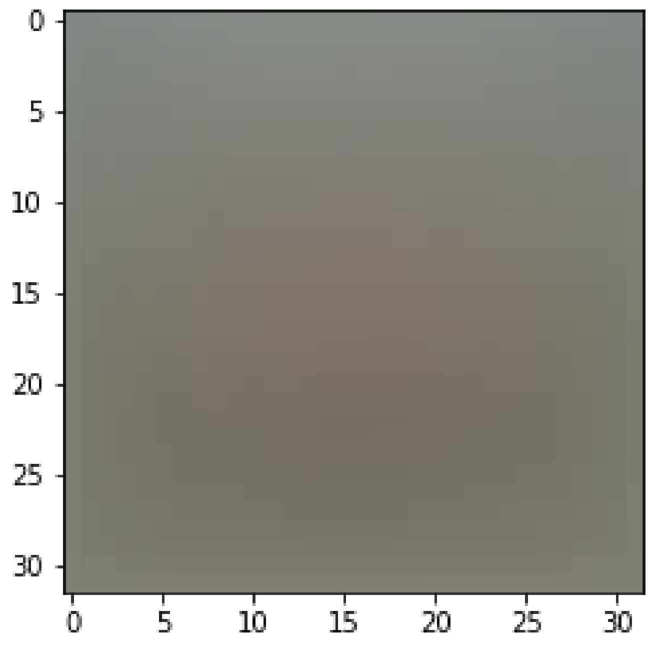
```
In [9]: # Preprocessing: reshape the image data into rows
        X_train = np.reshape(X_train, (X_train.shape[0], -1))
        X_val = np.reshape(X_val, (X_val.shape[0], -1))
        X_test = np.reshape(X_test, (X_test.shape[0], -1))
        X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

        # 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 [10]: # Preprocessing: subtract the mean image
         # first: compute the image mean based on the training data
         mean_image = np.mean(X_train, axis=0)
```

```
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

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

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

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



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

1.2 SVM Classifier

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

As you can see, we have prefilled the function `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

```
In [11]: # 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: 8.972882
```

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

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

```
In [12]: # Once you've implemented the gradient, recompute it with the code below
         # and gradient check it with the function we provided for you

         # 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: -25.960639 analytic: -25.960639, relative error: 1.125010e-11
numerical: 21.318670 analytic: 21.318670, relative error: 3.688337e-12
numerical: 8.408324 analytic: 8.408324, relative error: 2.993684e-11
numerical: 11.156094 analytic: 11.156094, relative error: 5.074899e-12
```

```
numerical: -21.260488 analytic: -21.260488, relative error: 4.389740e-12
numerical: 18.116979 analytic: 18.116979, relative error: 1.791557e-12
numerical: -13.680607 analytic: -13.680607, relative error: 2.379687e-11
numerical: 2.269483 analytic: 2.269483, relative error: 2.613183e-10
numerical: 12.214677 analytic: 12.214677, relative error: 2.999541e-11
numerical: 5.080400 analytic: 5.080400, relative error: 2.936536e-11
numerical: 3.773859 analytic: 3.773859, relative error: 5.347157e-11
numerical: -0.552298 analytic: -0.552298, relative error: 3.317093e-11
numerical: -7.374066 analytic: -7.374066, relative error: 3.950137e-11
numerical: -28.437498 analytic: -28.437498, relative error: 2.611549e-12
numerical: -19.818940 analytic: -19.818940, relative error: 8.502141e-12
numerical: -3.484888 analytic: -3.484888, relative error: 4.576629e-11
numerical: 13.164421 analytic: 13.164421, relative error: 5.287205e-12
numerical: 4.359284 analytic: 4.359284, relative error: 2.945437e-11
numerical: 14.926399 analytic: 14.926399, relative error: 1.996575e-11
numerical: -2.873812 analytic: -2.873812, relative error: 3.440526e-11
```

Inline Question 1

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

Your Answer : SVM's hingle loss function isn't defined at $|X| = 0$. Hence sometimes, the gradient check would fail.

```
In [13]: # Next implement the function svm_loss_vectorized; for now only compute the loss;
         # we will implement the gradient in a moment.
         tic = time.time()
         loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

         from cs231n.classifiers.linear_svm import svm_loss_vectorized
         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: 8.972882e+00 computed in 0.430151s
Vectorized loss: 8.972882e+00 computed in 0.032787s
difference: -0.000000
```

```
In [14]: # Complete the implementation of svm_loss_vectorized, and compute the gradient
         # of the loss function in a vectorized way.
         # The naive implementation and the vectorized implementation should match, but
         # the vectorized version should still be much faster.
         tic = time.time()
         _, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('Naive loss and gradient: computed in %fs' % (toc - tic))

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

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

```
Naive loss and gradient: computed in 0.634906s
Vectorized loss and gradient: computed in 0.009999s
difference: 0.000000
```

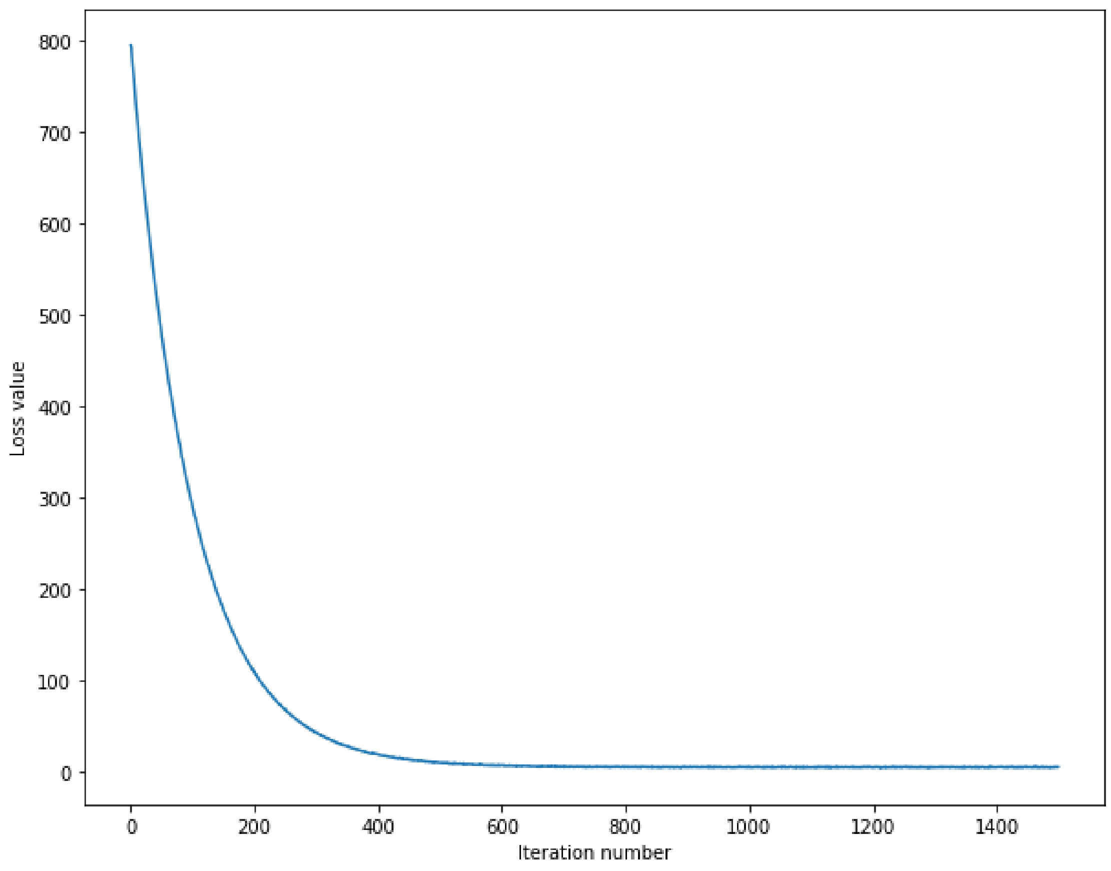
1.2.1 Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

```
In [15]: # In the file linear_classifier.py, implement SGD in the function
         # LinearClassifier.train() and then run it with the code below.
         from cs231n.classifiers import LinearSVM
         svm = LinearSVM()
         tic = time.time()
         loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                               num_iters=1500, verbose=True)
         toc = time.time()
         print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 794.874550
iteration 100 / 1500: loss 289.061102
iteration 200 / 1500: loss 109.891384
iteration 300 / 1500: loss 43.240302
iteration 400 / 1500: loss 19.134963
iteration 500 / 1500: loss 10.047260
iteration 600 / 1500: loss 7.628708
iteration 700 / 1500: loss 6.038818
iteration 800 / 1500: loss 5.462347
iteration 900 / 1500: loss 5.019107
iteration 1000 / 1500: loss 5.132940
iteration 1100 / 1500: loss 5.249205
iteration 1200 / 1500: loss 5.452096
iteration 1300 / 1500: loss 4.992620
iteration 1400 / 1500: loss 5.400390
That took 23.598874s
```

```
In [16]: # A useful debugging strategy is to plot the loss as a function of
         # iteration number:
         plt.plot(loss_hist)
         plt.xlabel('Iteration number')
         plt.ylabel('Loss value')
         plt.show()
```



```
In [17]: # Write the LinearSVM.predict function and evaluate the performance on both the
         # training and validation set
         y_train_pred = svm.predict(X_train)
         print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
```

```
        y_val_pred = svm.predict(X_val)
        print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

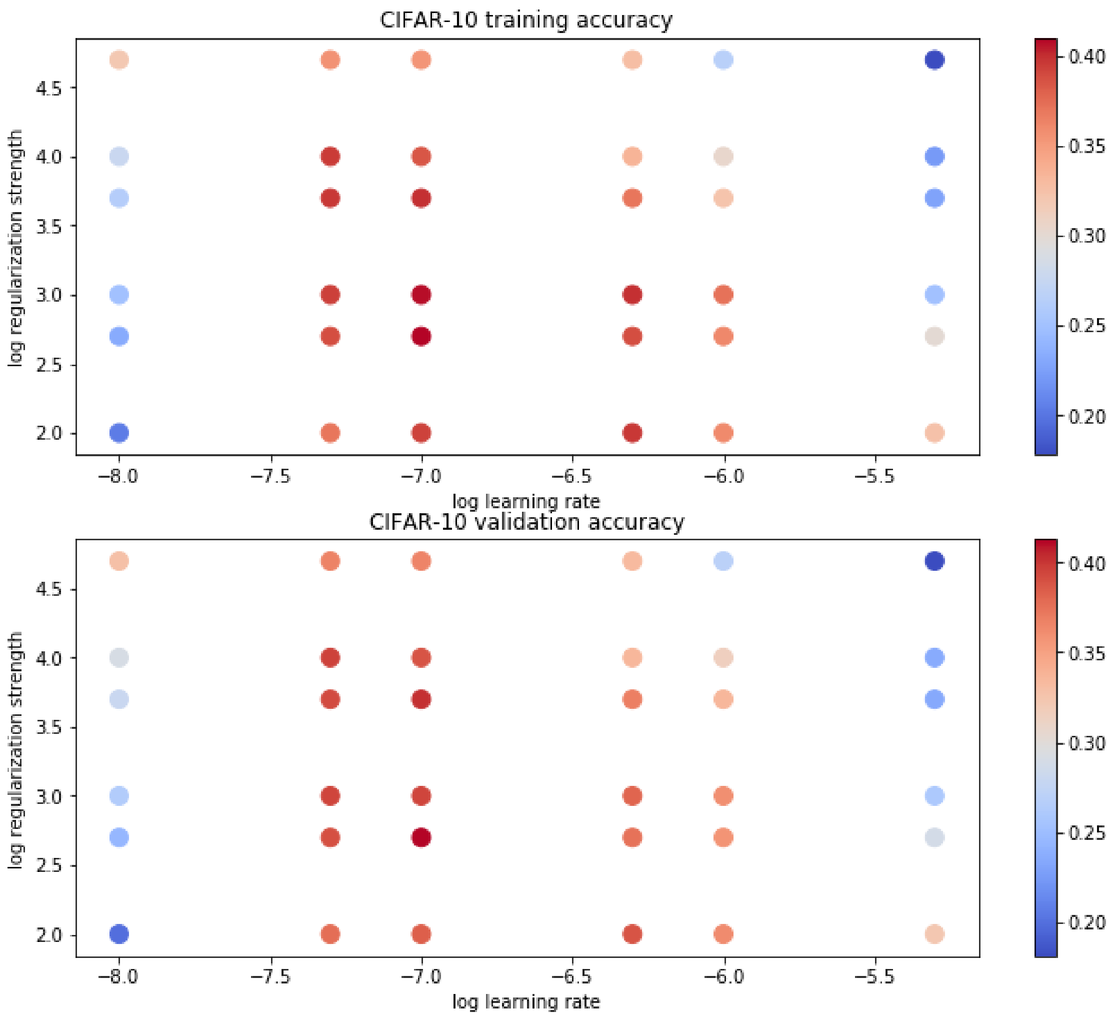
```
training accuracy: 0.371776
validation accuracy: 0.376000
```

```
In [20]: # Use the validation set to tune hyperparameters (regularization strength and
         # learning rate). You should experiment with different ranges for the learning
         # rates and regularization strengths; if you are careful you should be able to
         # get a classification accuracy of about 0.39 on the validation set.

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

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

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

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

         learning_rates = [1e-8, 5e-8, 1e-7, 5e-7, 1e-6, 5e-6]
         regularization_strengths = [1e2, 5e2, 1e3, 5e3, 1e4, 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.                          #
```

```
################################################################################

classifier = LinearSVM()

for lr in learning_rates:
    for reg in regularization_strengths:
        classifier.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=1000)

        y_train_pred = classifier.predict(X_train)
        acc_train = np.mean(y_train == y_train_pred)

        y_val_pred = classifier.predict(X_val)
        acc_val = np.mean(y_val == y_val_pred)

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

        if acc_val > best_val:
            best_val = acc_val
            best_softmax = classifier

# *****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-08 reg 1.000000e+02 train accuracy: 0.203837 val accuracy: 0.200000
lr 1.000000e-08 reg 5.000000e+02 train accuracy: 0.232959 val accuracy: 0.245000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.249571 val accuracy: 0.264000
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.263020 val accuracy: 0.280000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.277592 val accuracy: 0.291000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.319776 val accuracy: 0.329000
lr 5.000000e-08 reg 1.000000e+02 train accuracy: 0.370633 val accuracy: 0.377000
lr 5.000000e-08 reg 5.000000e+02 train accuracy: 0.388612 val accuracy: 0.391000
lr 5.000000e-08 reg 1.000000e+03 train accuracy: 0.393837 val accuracy: 0.395000
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.397286 val accuracy: 0.393000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.396020 val accuracy: 0.396000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.356224 val accuracy: 0.367000
lr 1.000000e-07 reg 1.000000e+02 train accuracy: 0.393449 val accuracy: 0.384000
lr 1.000000e-07 reg 5.000000e+02 train accuracy: 0.410020 val accuracy: 0.413000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.407408 val accuracy: 0.395000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.399673 val accuracy: 0.403000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.385041 val accuracy: 0.389000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.353857 val accuracy: 0.366000
lr 5.000000e-07 reg 1.000000e+02 train accuracy: 0.396878 val accuracy: 0.389000
lr 5.000000e-07 reg 5.000000e+02 train accuracy: 0.388327 val accuracy: 0.375000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.400020 val accuracy: 0.382000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.370510 val accuracy: 0.369000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.334918 val accuracy: 0.336000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.327735 val accuracy: 0.334000
lr 1.000000e-06 reg 1.000000e+02 train accuracy: 0.361000 val accuracy: 0.363000
lr 1.000000e-06 reg 5.000000e+02 train accuracy: 0.361143 val accuracy: 0.358000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.372735 val accuracy: 0.361000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.323061 val accuracy: 0.336000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.304429 val accuracy: 0.316000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.265245 val accuracy: 0.270000
lr 5.000000e-06 reg 1.000000e+02 train accuracy: 0.324816 val accuracy: 0.324000
lr 5.000000e-06 reg 5.000000e+02 train accuracy: 0.299694 val accuracy: 0.288000
lr 5.000000e-06 reg 1.000000e+03 train accuracy: 0.250061 val accuracy: 0.260000
lr 5.000000e-06 reg 5.000000e+03 train accuracy: 0.227510 val accuracy: 0.235000
lr 5.000000e-06 reg 1.000000e+04 train accuracy: 0.222020 val accuracy: 0.236000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.177918 val accuracy: 0.181000
best validation accuracy achieved during cross-validation: 0.413000
```

```
In [21]: # Visualize the cross-validation results
         import math
         import pdb

         # pdb.set_trace()

         x_scatter = [math.log10(x[0]) for x in results]
         y_scatter = [math.log10(x[1]) for x in results]

         # plot training accuracy
         marker_size = 100
         colors = [results[x][0] for x in results]
         plt.subplot(2, 1, 1)
         plt.tight_layout(pad=3)
         plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
         plt.colorbar()
         plt.xlabel('log learning rate')
         plt.ylabel('log regularization strength')
         plt.title('CIFAR-10 training accuracy')

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

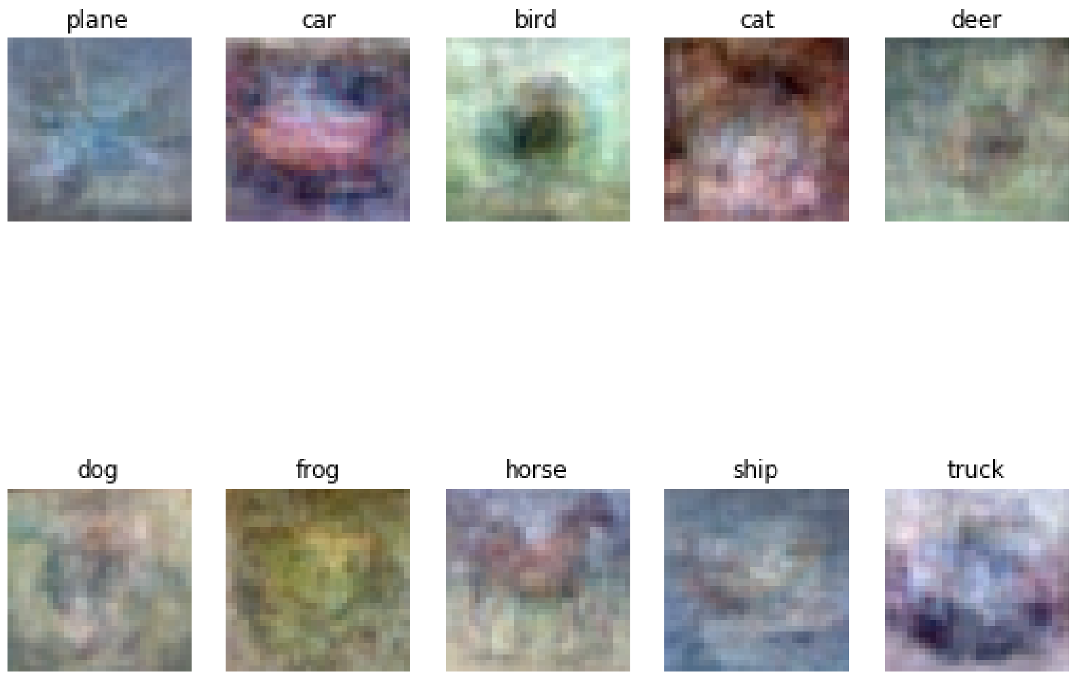


```
In [23]: # Evaluate the best svm on test set
         y_test_pred = best_softmax.predict(X_test)
         test_accuracy = np.mean(y_test == y_test_pred)
         print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.186000
```

```
In [25]: # 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_softmax.W[:-1,:] # strip out the bias
         w = w.reshape(32, 32, 3, 10)
```

```
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tr
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



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 weight visuaslisation give an insight on to what specific things in the image the classifier gets acitivated to. For a ship, we can see how a large intensity of blue color is there and for a cat we can see the two ears of the cat. Also, for the horse, the body shape is quite visible.

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.

2. Execute the cell below to download the modified `.py` files back to your drive.