svm

November 6, 2023

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

[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

[2]:
```python
# 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 issue)
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,)
```

[3]:
```python
# 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()
```

[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,)
```

[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)
```

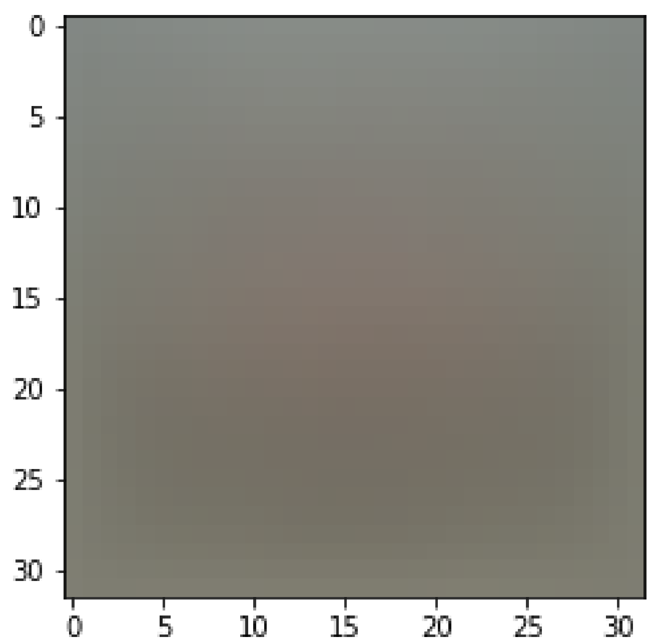
[6]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
 ↪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 `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

[7]:
```python
# 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: 8.712269
```

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:

[8]:
```python
# 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: 0.462022 analytic: 1.591270, relative error: 5.499699e-01
numerical: 2.822385 analytic: 1.612666, relative error: 2.727632e-01
numerical: -3.725127 analytic: -6.520000, relative error: 2.728002e-01
numerical: -3.233472 analytic: -4.223566, relative error: 1.327731e-01
numerical: 5.064060 analytic: 4.705114, relative error: 3.674269e-02
numerical: -42.718297 analytic: -44.802138, relative error: 2.380977e-02
numerical: -12.763390 analytic: -14.568000, relative error: 6.602702e-02
numerical: 23.120211 analytic: 23.912948, relative error: 1.685485e-02
numerical: -6.249832 analytic: -6.432484, relative error: 1.440215e-02
```

```
numerical: -0.560797 analytic: -2.197030, relative error: 5.933053e-01
numerical: -0.707724 analytic: -0.448628, relative error: 2.240633e-01
numerical: 12.242362 analytic: 13.634125, relative error: 5.378483e-02
numerical: -56.097077 analytic: -57.558069, relative error: 1.285460e-02
numerical: 19.472924 analytic: 22.113520, relative error: 6.349655e-02
numerical: 19.803707 analytic: 20.695416, relative error: 2.201798e-02
numerical: -0.257747 analytic: -0.106115, relative error: 4.167282e-01
numerical: 8.460260 analytic: 8.696922, relative error: 1.379378e-02
numerical: 6.528693 analytic: 6.917492, relative error: 2.891520e-02
numerical: -8.407918 analytic: -11.296405, relative error: 1.465915e-01
numerical: -6.053650 analytic: -0.679731, relative error: 7.981011e-01
```

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*

$\color{blue}{Your Answer:}$ *1.Yes, it is possible.2. because the right predictionclass brings the "0" gradient. 3. low down the margin and the numerical gradient would be accuracy.*

[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: 8.712269e+00 computed in 0.213207s
Vectorized loss: 8.712269e+00 computed in 0.015558s
difference: 0.000000
```

[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 0.214273s
Vectorized loss and gradient: computed in 0.008178s
difference: 287.772188
```

1.2.1 Stochastic Gradient Descent

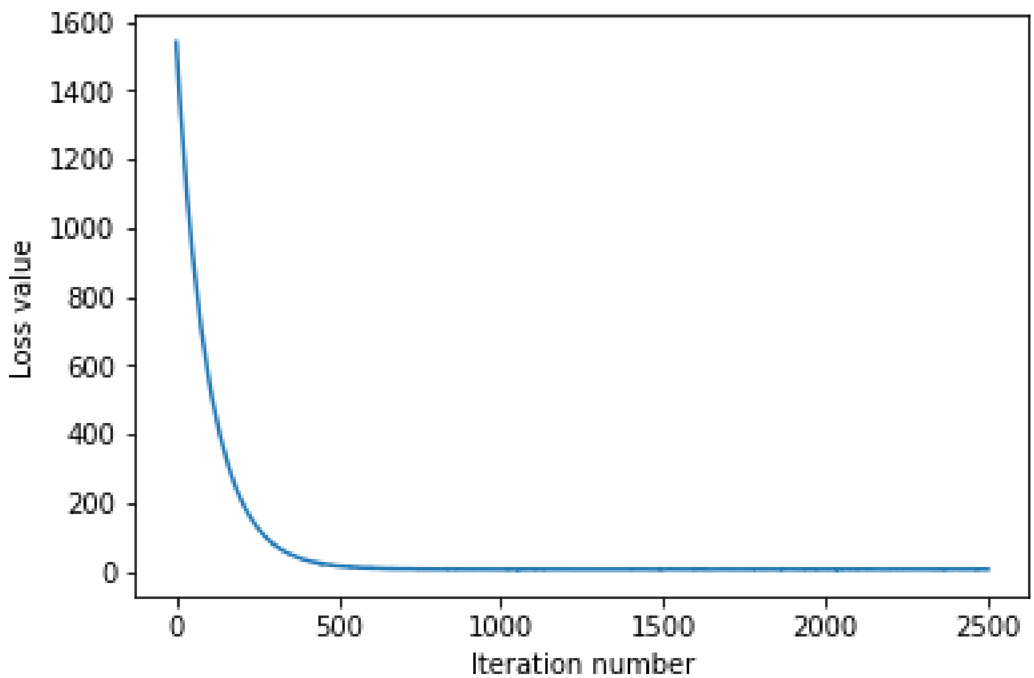
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[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=2500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 2500: loss 1539.965648
iteration 100 / 2500: loss 561.379683
iteration 200 / 2500: loss 207.386594
iteration 300 / 2500: loss 78.995212
iteration 400 / 2500: loss 32.351515
iteration 500 / 2500: loss 15.276761
iteration 600 / 2500: loss 8.815441
iteration 700 / 2500: loss 7.317820
iteration 800 / 2500: loss 6.649019
iteration 900 / 2500: loss 6.301109
iteration 1000 / 2500: loss 5.924322
iteration 1100 / 2500: loss 5.760378
iteration 1200 / 2500: loss 6.003845
```

```
iteration 1300 / 2500: loss 5.814065
iteration 1400 / 2500: loss 5.133910
iteration 1500 / 2500: loss 6.075364
iteration 1600 / 2500: loss 5.915600
iteration 1700 / 2500: loss 5.860075
iteration 1800 / 2500: loss 5.437388
iteration 1900 / 2500: loss 5.473663
iteration 2000 / 2500: loss 5.837305
iteration 2100 / 2500: loss 5.474469
iteration 2200 / 2500: loss 6.162901
iteration 2300 / 2500: loss 5.657615
iteration 2400 / 2500: loss 6.004385
That took 36.945611s
```

[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()
```



[17]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
```

```python
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: 0.373673
validation accuracy: 0.390000
```

[23]:

```python
# 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.                                      #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 2.7e-7, 5e-7]
regularization_strengths = [2.5e4,2.8e4, 3e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
    for reg in regularization_strengths:
```

```
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                  num_iters=2500, verbose=True)
        y_pre_val = svm.predict(X_val)
        acc_val = np.mean(y_pre_val == y_val)
        y_pre_train = svm.predict(X_train)
        acc_train = np.mean(y_pre_train == y_train)
        results[(lr, reg)] = (acc_train, acc_val)
        if acc_val > best_val:
            best_val = acc_val
            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)
```

```
iteration 0 / 2500: loss 1526.764108
iteration 100 / 2500: loss 555.903409
iteration 200 / 2500: loss 206.748601
iteration 300 / 2500: loss 78.707859
iteration 400 / 2500: loss 32.706547
iteration 500 / 2500: loss 15.322632
iteration 600 / 2500: loss 8.930914
iteration 700 / 2500: loss 6.704234
iteration 800 / 2500: loss 5.647990
iteration 900 / 2500: loss 5.768647
iteration 1000 / 2500: loss 5.984512
iteration 1100 / 2500: loss 5.759987
iteration 1200 / 2500: loss 5.655340
iteration 1300 / 2500: loss 5.893454
iteration 1400 / 2500: loss 5.268225
iteration 1500 / 2500: loss 5.531155
iteration 1600 / 2500: loss 5.315420
iteration 1700 / 2500: loss 5.838575
iteration 1800 / 2500: loss 6.029941
iteration 1900 / 2500: loss 6.637006
iteration 2000 / 2500: loss 5.721638
iteration 2100 / 2500: loss 6.276350
iteration 2200 / 2500: loss 5.776588
iteration 2300 / 2500: loss 5.540626
iteration 2400 / 2500: loss 6.238012
iteration 0 / 2500: loss 1727.438276
```

```
iteration 100 / 2500: loss 558.400934
iteration 200 / 2500: loss 183.303270
iteration 300 / 2500: loss 63.565720
iteration 400 / 2500: loss 23.564693
iteration 500 / 2500: loss 12.314244
iteration 600 / 2500: loss 7.406196
iteration 700 / 2500: loss 6.972662
iteration 800 / 2500: loss 6.282835
iteration 900 / 2500: loss 5.495134
iteration 1000 / 2500: loss 5.902560
iteration 1100 / 2500: loss 6.259144
iteration 1200 / 2500: loss 5.465933
iteration 1300 / 2500: loss 5.603724
iteration 1400 / 2500: loss 5.286954
iteration 1500 / 2500: loss 6.092411
iteration 1600 / 2500: loss 5.245575
iteration 1700 / 2500: loss 5.885536
iteration 1800 / 2500: loss 5.856265
iteration 1900 / 2500: loss 5.489966
iteration 2000 / 2500: loss 5.817674
iteration 2100 / 2500: loss 5.500146
iteration 2200 / 2500: loss 5.850321
iteration 2300 / 2500: loss 6.020091
iteration 2400 / 2500: loss 6.220059
iteration 0 / 2500: loss 1862.771412
iteration 100 / 2500: loss 557.241667
iteration 200 / 2500: loss 169.529050
iteration 300 / 2500: loss 54.828325
iteration 400 / 2500: loss 20.231291
iteration 500 / 2500: loss 10.133873
iteration 600 / 2500: loss 6.751382
iteration 700 / 2500: loss 6.104370
iteration 800 / 2500: loss 5.980064
iteration 900 / 2500: loss 6.753961
iteration 1000 / 2500: loss 5.733495
iteration 1100 / 2500: loss 5.894225
iteration 1200 / 2500: loss 5.600322
iteration 1300 / 2500: loss 5.582106
iteration 1400 / 2500: loss 5.546703
iteration 1500 / 2500: loss 5.595681
iteration 1600 / 2500: loss 5.657109
iteration 1700 / 2500: loss 5.790101
iteration 1800 / 2500: loss 5.808692
iteration 1900 / 2500: loss 5.782612
iteration 2000 / 2500: loss 6.070458
iteration 2100 / 2500: loss 5.438751
iteration 2200 / 2500: loss 5.624044
iteration 2300 / 2500: loss 5.413619
```

```
iteration 2400 / 2500: loss 6.263974
iteration 0 / 2500: loss 3087.615412
iteration 100 / 2500: loss 413.723687
iteration 200 / 2500: loss 60.228434
iteration 300 / 2500: loss 13.028564
iteration 400 / 2500: loss 7.461951
iteration 500 / 2500: loss 6.052723
iteration 600 / 2500: loss 6.002491
iteration 700 / 2500: loss 6.398235
iteration 800 / 2500: loss 6.250052
iteration 900 / 2500: loss 6.205412
iteration 1000 / 2500: loss 5.785935
iteration 1100 / 2500: loss 6.732863
iteration 1200 / 2500: loss 5.795034
iteration 1300 / 2500: loss 6.181659
iteration 1400 / 2500: loss 5.800003
iteration 1500 / 2500: loss 6.606022
iteration 1600 / 2500: loss 5.947812
iteration 1700 / 2500: loss 5.970044
iteration 1800 / 2500: loss 6.225508
iteration 1900 / 2500: loss 6.120258
iteration 2000 / 2500: loss 5.630256
iteration 2100 / 2500: loss 6.375773
iteration 2200 / 2500: loss 6.276251
iteration 2300 / 2500: loss 6.307940
iteration 2400 / 2500: loss 6.115284
iteration 0 / 2500: loss 1564.727854
iteration 100 / 2500: loss 105.349479
iteration 200 / 2500: loss 12.004695
iteration 300 / 2500: loss 6.290239
iteration 400 / 2500: loss 6.003306
iteration 500 / 2500: loss 5.751160
iteration 600 / 2500: loss 6.160714
iteration 700 / 2500: loss 6.225619
iteration 800 / 2500: loss 5.694208
iteration 900 / 2500: loss 6.295579
iteration 1000 / 2500: loss 5.996834
iteration 1100 / 2500: loss 5.734729
iteration 1200 / 2500: loss 5.736463
iteration 1300 / 2500: loss 6.338441
iteration 1400 / 2500: loss 6.390982
iteration 1500 / 2500: loss 5.824731
iteration 1600 / 2500: loss 6.109472
iteration 1700 / 2500: loss 6.224350
iteration 1800 / 2500: loss 5.943407
iteration 1900 / 2500: loss 6.133244
iteration 2000 / 2500: loss 5.871505
iteration 2100 / 2500: loss 5.726525
```

```
iteration 2200 / 2500: loss 5.390790
iteration 2300 / 2500: loss 5.889485
iteration 2400 / 2500: loss 6.022573
iteration 0 / 2500: loss 1729.905332
iteration 100 / 2500: loss 85.785315
iteration 200 / 2500: loss 9.322036
iteration 300 / 2500: loss 5.906160
iteration 400 / 2500: loss 5.672016
iteration 500 / 2500: loss 5.671883
iteration 600 / 2500: loss 5.920673
iteration 700 / 2500: loss 6.550629
iteration 800 / 2500: loss 5.751968
iteration 900 / 2500: loss 5.823675
iteration 1000 / 2500: loss 6.528528
iteration 1100 / 2500: loss 6.631678
iteration 1200 / 2500: loss 5.987235
iteration 1300 / 2500: loss 6.174091
iteration 1400 / 2500: loss 6.198019
iteration 1500 / 2500: loss 6.406457
iteration 1600 / 2500: loss 5.789751
iteration 1700 / 2500: loss 6.106288
iteration 1800 / 2500: loss 5.475280
iteration 1900 / 2500: loss 6.263491
iteration 2000 / 2500: loss 6.135025
iteration 2100 / 2500: loss 6.100293
iteration 2200 / 2500: loss 5.586282
iteration 2300 / 2500: loss 6.330452
iteration 2400 / 2500: loss 6.001261
iteration 0 / 2500: loss 1870.790240
iteration 100 / 2500: loss 75.355762
iteration 200 / 2500: loss 8.446794
iteration 300 / 2500: loss 6.153208
iteration 400 / 2500: loss 5.898120
iteration 500 / 2500: loss 5.740559
iteration 600 / 2500: loss 6.314207
iteration 700 / 2500: loss 6.129737
iteration 800 / 2500: loss 5.801667
iteration 900 / 2500: loss 6.064820
iteration 1000 / 2500: loss 6.937610
iteration 1100 / 2500: loss 6.401892
iteration 1200 / 2500: loss 5.500643
iteration 1300 / 2500: loss 6.114677
iteration 1400 / 2500: loss 5.952831
iteration 1500 / 2500: loss 5.692656
iteration 1600 / 2500: loss 5.854318
iteration 1700 / 2500: loss 5.860182
iteration 1800 / 2500: loss 5.982419
iteration 1900 / 2500: loss 6.004835
```

```
iteration 2000 / 2500: loss 6.126762
iteration 2100 / 2500: loss 5.143830
iteration 2200 / 2500: loss 6.128610
iteration 2300 / 2500: loss 6.063196
iteration 2400 / 2500: loss 6.779481
iteration 0 / 2500: loss 3087.924943
iteration 100 / 2500: loss 18.985306
iteration 200 / 2500: loss 6.154246
iteration 300 / 2500: loss 6.350338
iteration 400 / 2500: loss 6.368004
iteration 500 / 2500: loss 6.572740
iteration 600 / 2500: loss 6.427202
iteration 700 / 2500: loss 6.562124
iteration 800 / 2500: loss 6.262650
iteration 900 / 2500: loss 6.714682
iteration 1000 / 2500: loss 6.452524
iteration 1100 / 2500: loss 6.203776
iteration 1200 / 2500: loss 6.801486
iteration 1300 / 2500: loss 6.879437
iteration 1400 / 2500: loss 6.443098
iteration 1500 / 2500: loss 6.260388
iteration 1600 / 2500: loss 6.782349
iteration 1700 / 2500: loss 6.603363
iteration 1800 / 2500: loss 6.400724
iteration 1900 / 2500: loss 6.708523
iteration 2000 / 2500: loss 6.413788
iteration 2100 / 2500: loss 6.512039
iteration 2200 / 2500: loss 6.903329
iteration 2300 / 2500: loss 6.283869
iteration 2400 / 2500: loss 6.502541
iteration 0 / 2500: loss 1556.910936
iteration 100 / 2500: loss 15.545196
iteration 200 / 2500: loss 6.847975
iteration 300 / 2500: loss 6.496135
iteration 400 / 2500: loss 5.947165
iteration 500 / 2500: loss 5.839728
iteration 600 / 2500: loss 7.476450
iteration 700 / 2500: loss 5.900250
iteration 800 / 2500: loss 6.802623
iteration 900 / 2500: loss 6.571946
iteration 1000 / 2500: loss 5.809760
iteration 1100 / 2500: loss 6.195540
iteration 1200 / 2500: loss 6.044200
iteration 1300 / 2500: loss 5.753824
iteration 1400 / 2500: loss 7.276126
iteration 1500 / 2500: loss 5.696942
iteration 1600 / 2500: loss 5.725970
iteration 1700 / 2500: loss 6.405830
```

```
iteration 1800 / 2500: loss 6.305769
iteration 1900 / 2500: loss 6.447331
iteration 2000 / 2500: loss 5.967933
iteration 2100 / 2500: loss 6.517507
iteration 2200 / 2500: loss 5.934676
iteration 2300 / 2500: loss 6.386009
iteration 2400 / 2500: loss 6.285869
iteration 0 / 2500: loss 1735.403853
iteration 100 / 2500: loss 12.109051
iteration 200 / 2500: loss 6.475169
iteration 300 / 2500: loss 6.651912
iteration 400 / 2500: loss 5.904546
iteration 500 / 2500: loss 6.952874
iteration 600 / 2500: loss 5.874488
iteration 700 / 2500: loss 6.614702
iteration 800 / 2500: loss 6.276269
iteration 900 / 2500: loss 6.622174
iteration 1000 / 2500: loss 5.834377
iteration 1100 / 2500: loss 6.551499
iteration 1200 / 2500: loss 6.541336
iteration 1300 / 2500: loss 6.182718
iteration 1400 / 2500: loss 6.719306
iteration 1500 / 2500: loss 6.478988
iteration 1600 / 2500: loss 6.408590
iteration 1700 / 2500: loss 6.558326
iteration 1800 / 2500: loss 6.477376
iteration 1900 / 2500: loss 6.312322
iteration 2000 / 2500: loss 6.493144
iteration 2100 / 2500: loss 5.872902
iteration 2200 / 2500: loss 5.963154
iteration 2300 / 2500: loss 6.537677
iteration 2400 / 2500: loss 7.017686
iteration 0 / 2500: loss 1850.733208
iteration 100 / 2500: loss 10.398369
iteration 200 / 2500: loss 6.677977
iteration 300 / 2500: loss 5.918756
iteration 400 / 2500: loss 5.959824
iteration 500 / 2500: loss 6.510411
iteration 600 / 2500: loss 6.144874
iteration 700 / 2500: loss 6.140353
iteration 800 / 2500: loss 6.663407
iteration 900 / 2500: loss 6.511005
iteration 1000 / 2500: loss 6.789212
iteration 1100 / 2500: loss 5.909318
iteration 1200 / 2500: loss 6.480827
iteration 1300 / 2500: loss 5.943624
iteration 1400 / 2500: loss 6.190863
iteration 1500 / 2500: loss 6.531566
```

```
iteration 1600 / 2500: loss 6.453937
iteration 1700 / 2500: loss 6.647823
iteration 1800 / 2500: loss 6.761740
iteration 1900 / 2500: loss 6.847304
iteration 2000 / 2500: loss 6.152155
iteration 2100 / 2500: loss 6.570023
iteration 2200 / 2500: loss 6.242002
iteration 2300 / 2500: loss 6.349284
iteration 2400 / 2500: loss 6.451821
iteration 0 / 2500: loss 3095.042501
iteration 100 / 2500: loss 6.634344
iteration 200 / 2500: loss 6.739679
iteration 300 / 2500: loss 6.988874
iteration 400 / 2500: loss 7.367875
iteration 500 / 2500: loss 7.377774
iteration 600 / 2500: loss 6.802317
iteration 700 / 2500: loss 6.756541
iteration 800 / 2500: loss 7.151827
iteration 900 / 2500: loss 6.375207
iteration 1000 / 2500: loss 6.520180
iteration 1100 / 2500: loss 6.809941
iteration 1200 / 2500: loss 6.709783
iteration 1300 / 2500: loss 6.924267
iteration 1400 / 2500: loss 6.812407
iteration 1500 / 2500: loss 6.346697
iteration 1600 / 2500: loss 6.557444
iteration 1700 / 2500: loss 7.078765
iteration 1800 / 2500: loss 6.809614
iteration 1900 / 2500: loss 6.679415
iteration 2000 / 2500: loss 6.725176
iteration 2100 / 2500: loss 6.537507
iteration 2200 / 2500: loss 7.348272
iteration 2300 / 2500: loss 6.686828
iteration 2400 / 2500: loss 6.489666
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.371408 val accuracy: 0.382000
lr 1.000000e-07 reg 2.800000e+04 train accuracy: 0.364449 val accuracy: 0.380000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.366939 val accuracy: 0.372000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.356143 val accuracy: 0.354000
lr 2.700000e-07 reg 2.500000e+04 train accuracy: 0.357694 val accuracy: 0.358000
lr 2.700000e-07 reg 2.800000e+04 train accuracy: 0.346041 val accuracy: 0.356000
lr 2.700000e-07 reg 3.000000e+04 train accuracy: 0.358796 val accuracy: 0.362000
lr 2.700000e-07 reg 5.000000e+04 train accuracy: 0.338224 val accuracy: 0.348000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.322367 val accuracy: 0.322000
lr 5.000000e-07 reg 2.800000e+04 train accuracy: 0.322694 val accuracy: 0.333000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.336857 val accuracy: 0.343000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.301633 val accuracy: 0.317000
best validation accuracy achieved during cross-validation: 0.382000
```

[ ]:
```
# 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()
```

[ ]:
```
# 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)
```

[ ]:
```
# 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', '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])
```

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.

$\color{blue}{Your Answer:}$ *The Description: the weight looks like a blur photo of each class and it's also like the combination of each photos.The interpretation: The shape of the image content looks like the original object of the class because the shape of original class has some feature of it. the weight just combine them together.*