svm

September 29, 2023

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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

[2]:
```python
# 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
```

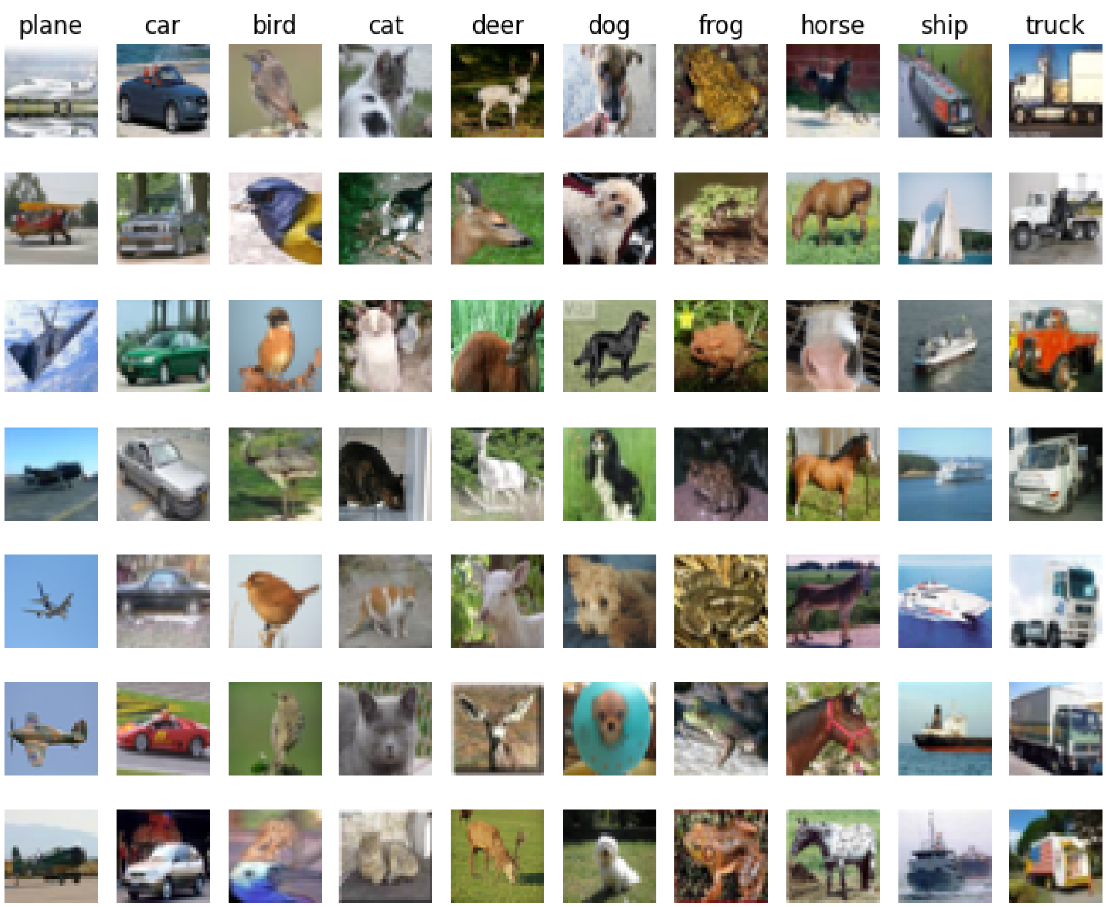
1.1 CIFAR-10 Data Loading and Preprocessing

[3]:
```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,)
```

[4]:

```
# 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()
```



[5]:
```python
# 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)
```

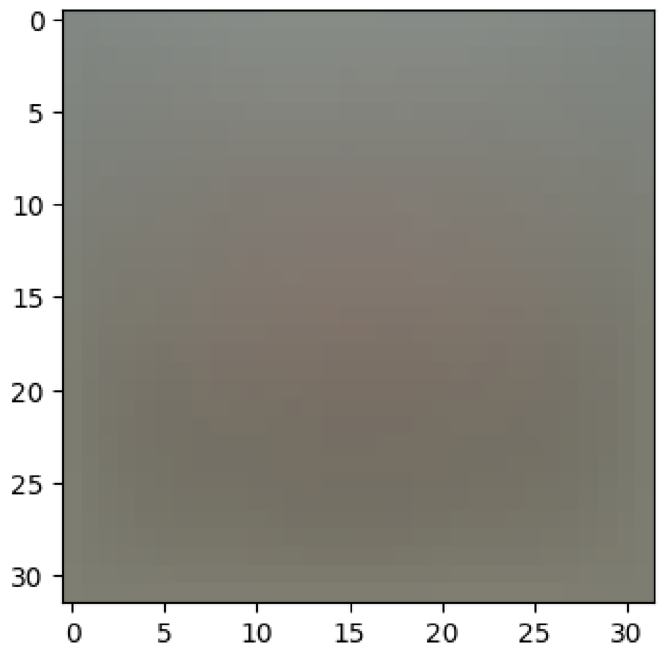
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[6]:
```python
# 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)
```

[7]:
```python
# 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.

[8]:
```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: 8.967418
```

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:

[9]:
```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: 11.042283 analytic: 0.000000, relative error: 1.000000e+00
numerical: 1.699520 analytic: 0.000000, relative error: 1.000000e+00
numerical: 6.855507 analytic: 0.000000, relative error: 1.000000e+00
numerical: -0.912618 analytic: 0.000000, relative error: 1.000000e+00
numerical: 9.911521 analytic: 0.000000, relative error: 1.000000e+00
numerical: 3.268000 analytic: 0.000000, relative error: 1.000000e+00
numerical: -10.220423 analytic: 0.000000, relative error: 1.000000e+00
numerical: -2.412000 analytic: 0.000000, relative error: 1.000000e+00
numerical: -6.367687 analytic: 0.000000, relative error: 1.000000e+00
numerical: -4.606000 analytic: 0.000000, relative error: 1.000000e+00
numerical: -14.004909 analytic: -0.001762, relative error: 9.997484e-01
numerical: -11.582655 analytic: 0.006174, relative error: 1.000000e+00
numerical: 8.354130 analytic: 0.002433, relative error: 9.994178e-01
numerical: -20.889944 analytic: 0.007170, relative error: 1.000000e+00
numerical: 16.217820 analytic: -0.000159, relative error: 1.000000e+00
numerical: -1.190195 analytic: 0.005412, relative error: 1.000000e+00
numerical: 1.330341 analytic: 0.011780, relative error: 9.824464e-01
numerical: 15.946821 analytic: -0.003691, relative error: 1.000000e+00
numerical: -8.981715 analytic: -0.004985, relative error: 9.988905e-01
numerical: -6.708374 analytic: -0.010764, relative error: 9.967959e-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*

$Your Answer:$ The SVM loss function is defined as the maximum between 0 and the score value subtracted from the margin, and it is not differentiable precisely at hinge point where margin equals 1. During gradient checks, it is expected that the analytical gradient and numerically approximated gradient may not match exactly.

To check if differentiation fails at the hinge point, both analytical and numerical methods can be used. The error between the two gradients is typically very small but not exactly zero.

Increasing the safety margin to avoid the exact hinge point (e.g., using margin of 1.01 instead of 1) could potentially mitigate the issue, but it comes at the cost of allowing more margin violations, which might lead to lower classification accuracy. Thus, there is a trade-off between achieving a smoother loss surface and maintaining the classification performance.

[10]:
```
# 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: 8.967418e+00 computed in 0.018630s
Vectorized loss: 8.967418e+00 computed in 0.013200s
difference: -0.000000
```

[11]:
```
# 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 0.029247s
Vectorized loss and gradient: computed in 0.012363s
difference: 3040.613243
```

1.2.1 Stochastic Gradient Descent

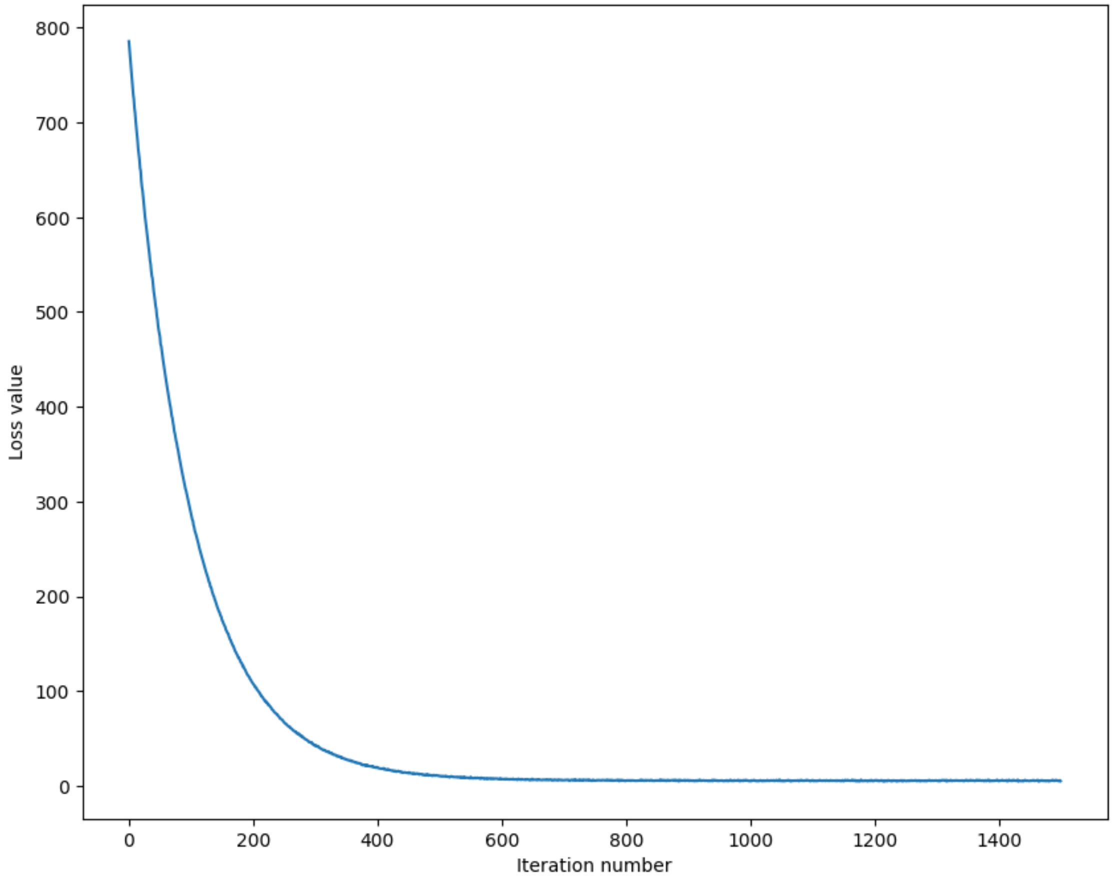
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[12]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 785.672391
iteration 100 / 1500: loss 287.056242
iteration 200 / 1500: loss 107.367213
iteration 300 / 1500: loss 42.321357
iteration 400 / 1500: loss 19.346025
iteration 500 / 1500: loss 10.252051
iteration 600 / 1500: loss 6.860802
iteration 700 / 1500: loss 5.715234
iteration 800 / 1500: loss 5.439343
iteration 900 / 1500: loss 5.836427
iteration 1000 / 1500: loss 5.087377
iteration 1100 / 1500: loss 5.760094
iteration 1200 / 1500: loss 5.525966
iteration 1300 / 1500: loss 5.303209
iteration 1400 / 1500: loss 5.129233
That took 11.860566s
```

[13]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
```

```
plt.ylabel('Loss value')
plt.show()
```



[14]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.367184
validation accuracy: 0.372000
```

[15]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
```

```python
# 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, 2e-7, 5e-8]
regularization_strengths = [2e4, 5e4]

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

# To find the optimal lr and reg strength we have to create 2 loops to figure␣
 ↪out the best hyperparameters to find the best softmax classifier.
best_learning_rate = 0
best_reg_strength = 0
# Looping over all combinations of hyperparameters
for lr in learning_rates:
    for reg in regularization_strengths:
        # Create a LinearSVM classifier object with the current hyperparameters
        svm = LinearSVM()

        # Training the classifier on the training data with 1200 iterations
        svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                  num_iters=1200, verbose=True)
```

```python
        # Prediction on the training and validation sets
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)

        # Computing accuracy on training and validation sets
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        # Storing the accuracy values in the results dictionary
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # Checking if this is the best validation accuracy so far and adding to
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm
            best_learning_rate = lr
            best_reg_strength = reg
print(f"Best Pair of lr and reg is (lr,reg):
 ↪{(best_learning_rate,best_reg_strength)}")

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)
```

```
iteration 0 / 1200: loss 636.392334
iteration 100 / 1200: loss 282.463443
iteration 200 / 1200: loss 128.486229
iteration 300 / 1200: loss 60.529248
iteration 400 / 1200: loss 29.589954
iteration 500 / 1200: loss 15.707261
iteration 600 / 1200: loss 10.195858
iteration 700 / 1200: loss 7.366290
iteration 800 / 1200: loss 6.459745
iteration 900 / 1200: loss 5.428998
iteration 1000 / 1200: loss 5.239562
iteration 1100 / 1200: loss 5.415083
iteration 0 / 1200: loss 1552.683104
iteration 100 / 1200: loss 209.470476
```

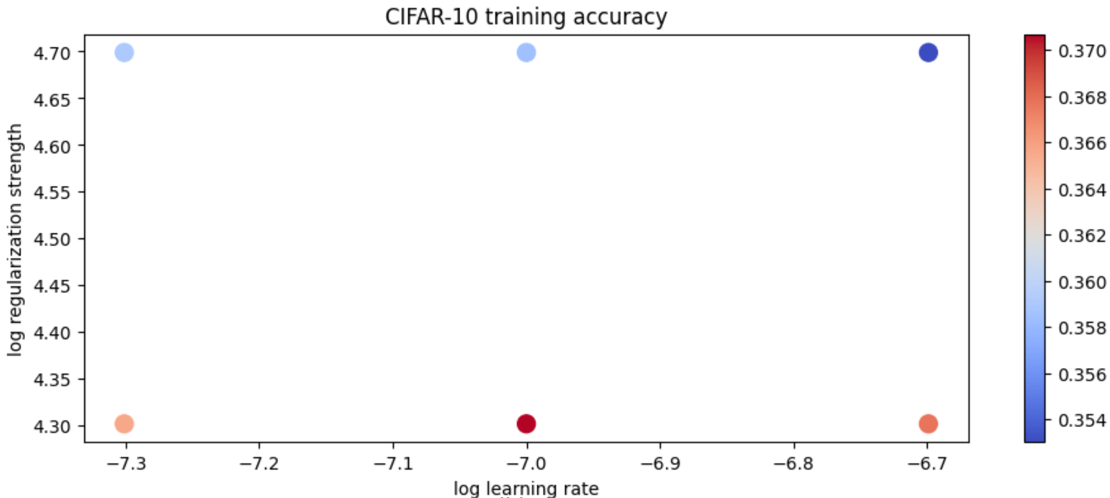
```
iteration 200 / 1200: loss 32.854935
iteration 300 / 1200: loss 9.208821
iteration 400 / 1200: loss 6.303889
iteration 500 / 1200: loss 5.818498
iteration 600 / 1200: loss 5.472783
iteration 700 / 1200: loss 6.092746
iteration 800 / 1200: loss 5.401369
iteration 900 / 1200: loss 5.370357
iteration 1000 / 1200: loss 5.777847
iteration 1100 / 1200: loss 5.929584
iteration 0 / 1200: loss 635.229452
iteration 100 / 1200: loss 127.916151
iteration 200 / 1200: loss 30.352040
iteration 300 / 1200: loss 9.911900
iteration 400 / 1200: loss 6.132874
iteration 500 / 1200: loss 5.532101
iteration 600 / 1200: loss 5.743142
iteration 700 / 1200: loss 5.538841
iteration 800 / 1200: loss 4.813722
iteration 900 / 1200: loss 5.588044
iteration 1000 / 1200: loss 4.998336
iteration 1100 / 1200: loss 5.170718
iteration 0 / 1200: loss 1564.782100
iteration 100 / 1200: loss 32.586659
iteration 200 / 1200: loss 6.038992
iteration 300 / 1200: loss 5.853585
iteration 400 / 1200: loss 6.455551
iteration 500 / 1200: loss 5.598117
iteration 600 / 1200: loss 5.722990
iteration 700 / 1200: loss 5.473484
iteration 800 / 1200: loss 6.258077
iteration 900 / 1200: loss 6.151456
iteration 1000 / 1200: loss 5.568236
iteration 1100 / 1200: loss 5.370822
iteration 0 / 1200: loss 627.550705
iteration 100 / 1200: loss 417.789850
iteration 200 / 1200: loss 279.135987
iteration 300 / 1200: loss 188.053498
iteration 400 / 1200: loss 127.587273
iteration 500 / 1200: loss 87.013248
iteration 600 / 1200: loss 60.033360
iteration 700 / 1200: loss 41.272632
iteration 800 / 1200: loss 29.480405
iteration 900 / 1200: loss 21.738501
iteration 1000 / 1200: loss 15.624555
iteration 1100 / 1200: loss 12.226743
iteration 0 / 1200: loss 1546.822016
iteration 100 / 1200: loss 565.487708
```

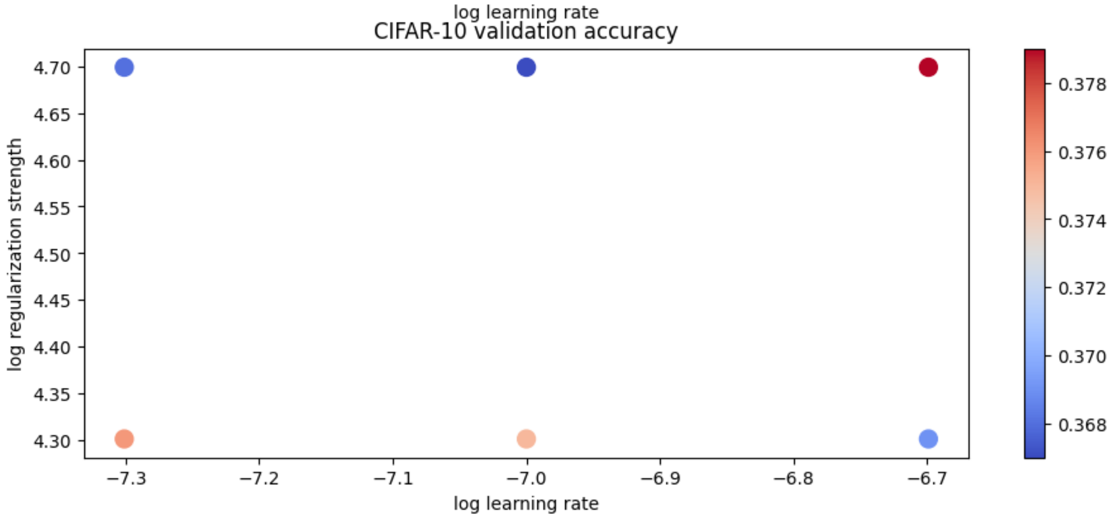
```
iteration 200 / 1200: loss 209.495753
iteration 300 / 1200: loss 79.311019
iteration 400 / 1200: loss 32.432737
iteration 500 / 1200: loss 15.301361
iteration 600 / 1200: loss 8.676943
iteration 700 / 1200: loss 7.023456
iteration 800 / 1200: loss 5.886734
iteration 900 / 1200: loss 5.687791
iteration 1000 / 1200: loss 5.454899
iteration 1100 / 1200: loss 5.642962
Best Pair of lr and reg is (lr,reg):(2e-07, 50000.0)
lr 5.000000e-08 reg 2.000000e+04 train accuracy: 0.365633 val accuracy: 0.376000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.359061 val accuracy: 0.368000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.370694 val accuracy: 0.375000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.358408 val accuracy: 0.367000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.367755 val accuracy: 0.369000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.353000 val accuracy: 0.379000
best validation accuracy achieved during cross-validation: 0.379000
```

[16]:

```python
# 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()
```



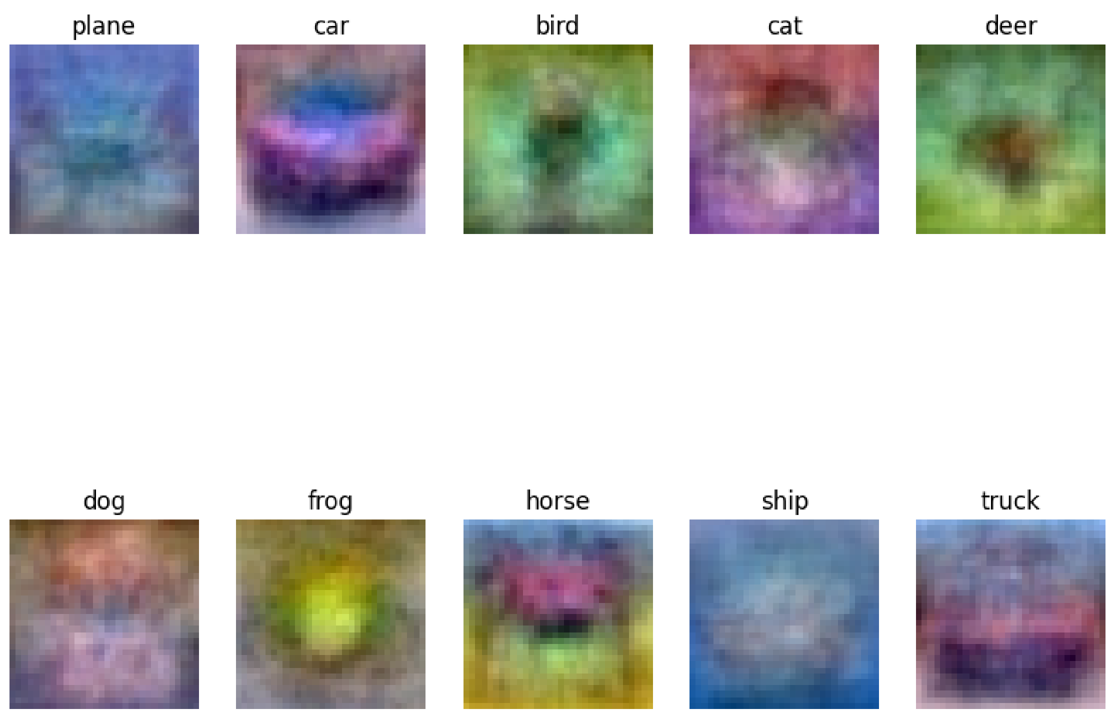


[17]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.365000
```

[18]:
```
# 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)
```

```python
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.

YourAnswer : The visualization of SVM weights typically resembles a set of color maps or images, where each pixel corresponds to a feature in the input space. The color or intensity of each pixel represents the weight associated with that feature. These weights capture the importance of each feature in making classification decisions.

For exampkle, if we're using an SVM to classify images of objects, such as cars, the SVM weights for the "car" class might exhibit patterns resembling a car's front view. This is because the SVM has learned that specific features, like the shape of the car's headlights or grille, are significant for identifying cars in the dataset.

In essence, SVM weights visually show the distinctive characteristics and patterns in the training data that are crucial for distinguishing between different classes. They reflect what the SVM has learned about the important features that contribute to its classification decisions.