svm

January 21, 2023

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

# TODO: 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh

%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/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

[53]:
```
# 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
```

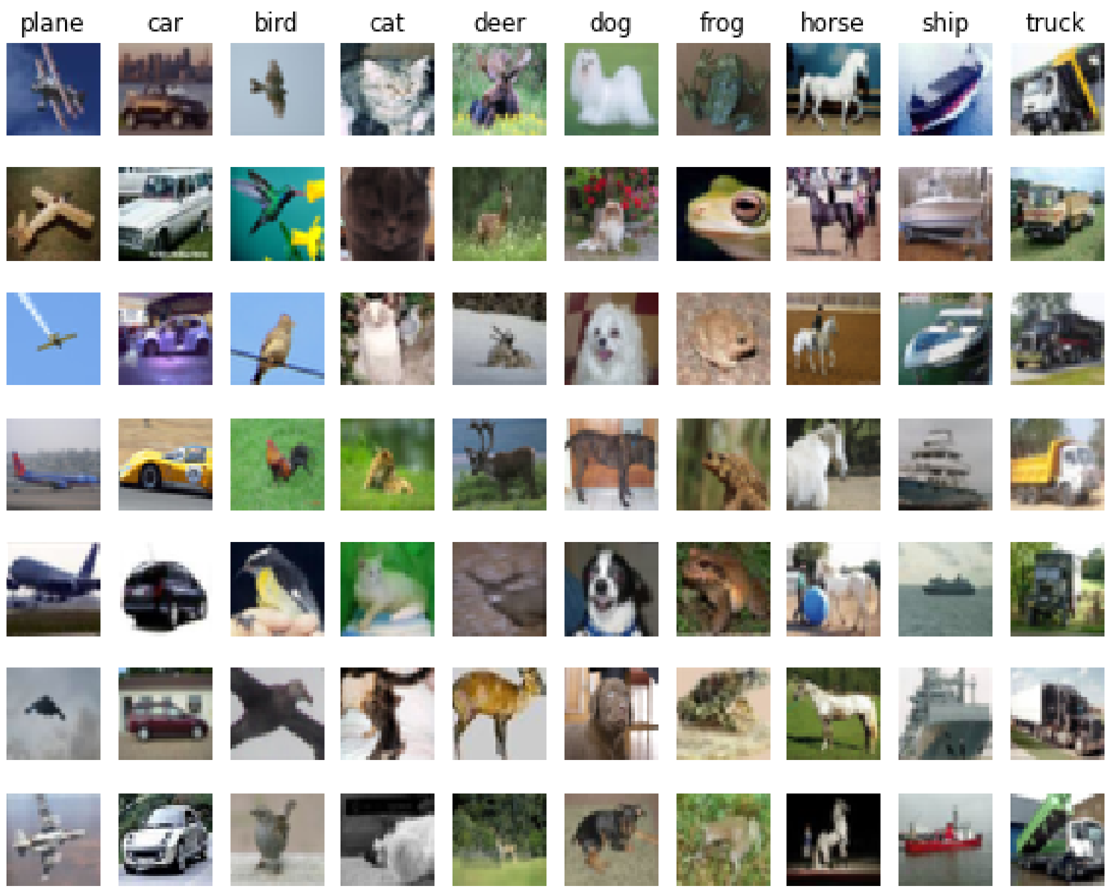
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 CIFAR-10 Data Loading and Preprocessing

[54]:
```
# 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)
```

```
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,)
```

[55]:
```
# 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()
```



[56]:
```
# 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,)
```

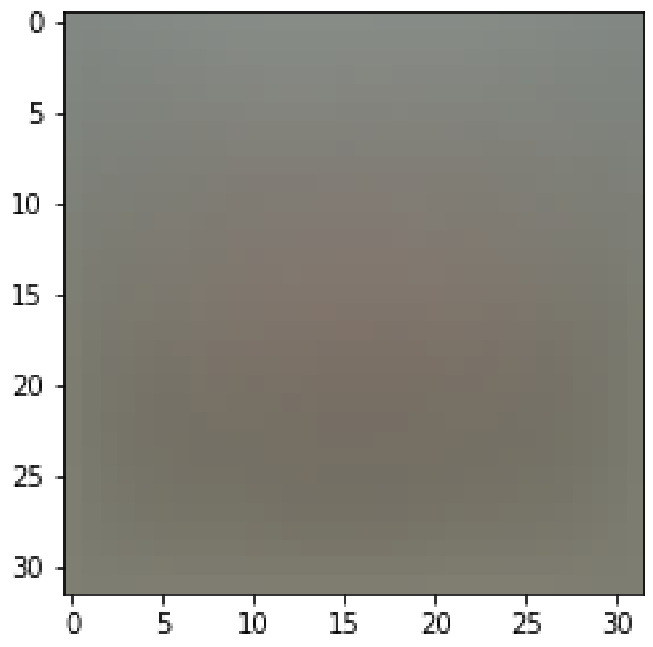
[57]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

[58]:
```
# 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.

[59]:
```
# 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: 9.032052
```

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, 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:

[60]:
```
# 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: 6.073638 analytic: 3.864498, relative error: 2.222892e-01
numerical: 16.610551 analytic: 18.940551, relative error: 6.553946e-02
numerical: -60.806648 analytic: -60.240937, relative error: 4.673455e-03
```

```
numerical: -21.323954 analytic: -21.403476, relative error: 1.861138e-03
numerical: 1.067460 analytic: 0.516201, relative error: 3.480918e-01
numerical: 13.780860 analytic: 17.057331, relative error: 1.062472e-01
numerical: -27.149476 analytic: -26.880184, relative error: 4.984139e-03
numerical: 0.412071 analytic: 2.256071, relative error: 6.911176e-01
numerical: -7.586915 analytic: -7.050364, relative error: 3.665645e-02
numerical: 9.505773 analytic: 9.636000, relative error: 6.803311e-03
numerical: -0.296842 analytic: -3.631157, relative error: 8.488582e-01
numerical: -7.690065 analytic: -8.456179, relative error: 4.744845e-02
numerical: 18.192310 analytic: 15.479330, relative error: 8.057166e-02
numerical: 1.784269 analytic: 0.806526, relative error: 3.773908e-01
numerical: 9.883469 analytic: 9.972486, relative error: 4.483108e-03
numerical: -14.132113 analytic: -13.594012, relative error: 1.940774e-02
numerical: -8.423281 analytic: -8.787009, relative error: 2.113436e-02
numerical: 0.654575 analytic: -1.214298, relative error: 1.000000e+00
numerical: -16.790212 analytic: -16.312683, relative error: 1.442560e-02
numerical: 24.797484 analytic: 23.487243, relative error: 2.713572e-02
```

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 : 1.Yes, it is possible . 2. because the right prediction class brings the "0" gradient. 3. low down the margin and the numerical gradient would be accuracy.

[61]:
```python
# 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: 9.032052e+00 computed in 0.144386s
Vectorized loss: 9.032052e+00 computed in 0.017440s
difference: -0.000000
```

[62]:
```
# 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(grad_naive.sum())
print(grad_vectorized.sum())
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.117815s
Vectorized loss and gradient: computed in 0.011723s
-3.040733531634032e-07
-3.041051499508285e-07
difference: 373.955287
```

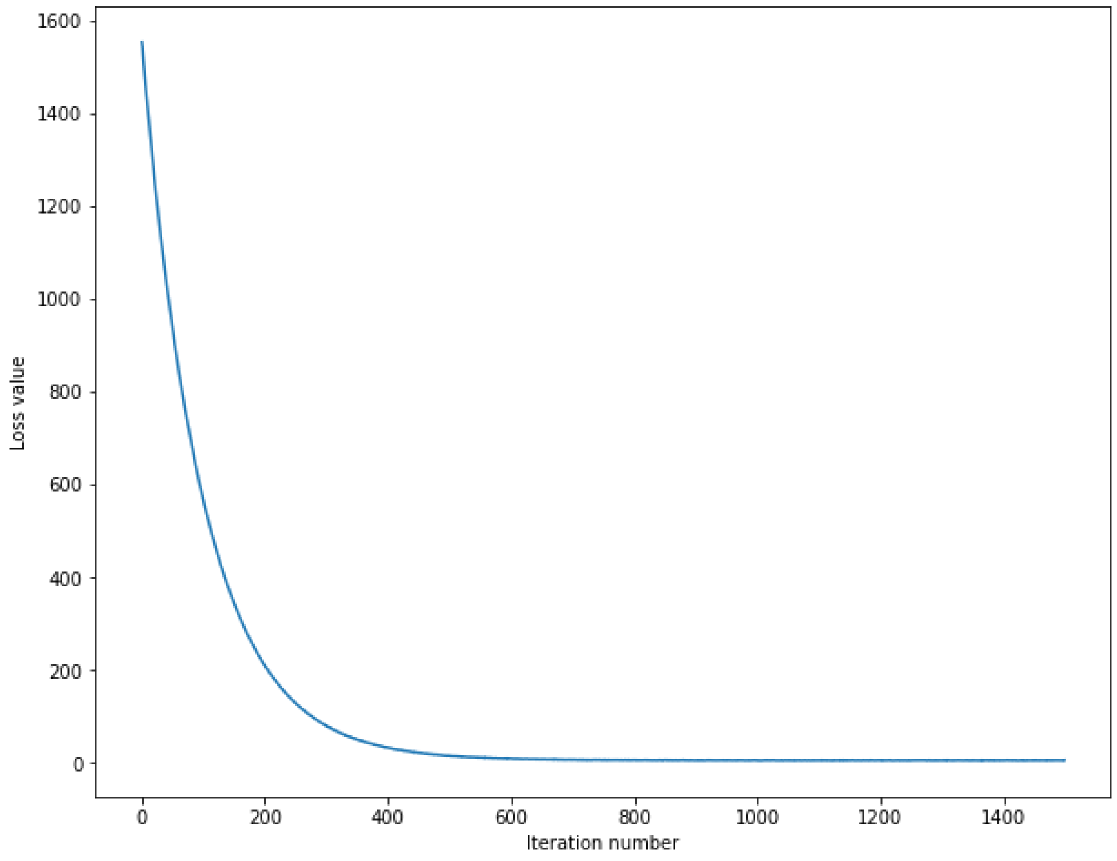
1.2.1 Stochastic Gradient Descent

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`.

[63]:
```
# 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 1554.213871
iteration 100 / 1500: loss 565.795905
iteration 200 / 1500: loss 209.649292
iteration 300 / 1500: loss 78.963692
iteration 400 / 1500: loss 32.863429
iteration 500 / 1500: loss 16.328531
iteration 600 / 1500: loss 9.354626
iteration 700 / 1500: loss 6.767091
iteration 800 / 1500: loss 6.285787
iteration 900 / 1500: loss 5.835103
iteration 1000 / 1500: loss 5.300083
iteration 1100 / 1500: loss 5.800378
iteration 1200 / 1500: loss 5.818707
iteration 1300 / 1500: loss 5.674870
iteration 1400 / 1500: loss 5.610138
That took 20.522027s
```

[64]:
```
# 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()
```



[65]:
```
# 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), ))
```

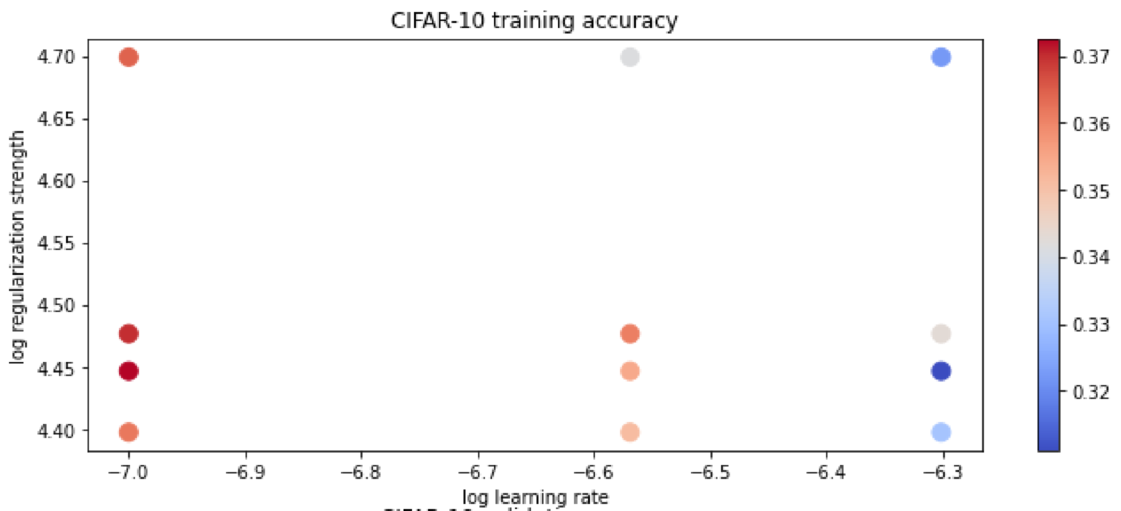
```
training accuracy: 0.368224
validation accuracy: 0.378000
```

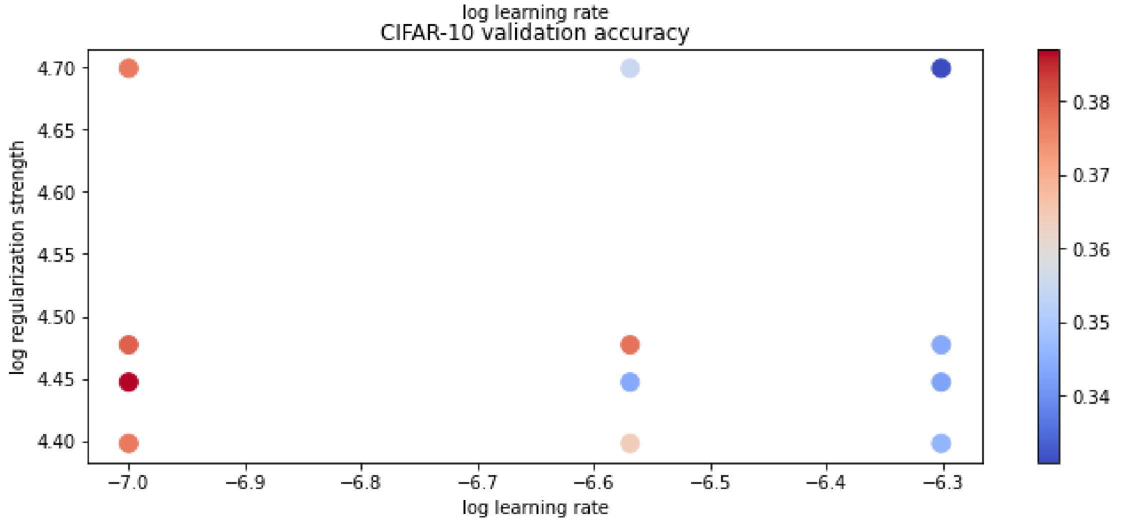
[72]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) 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=1000, 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 / 1000: loss 1550.740606
iteration 100 / 1000: loss 564.733450
iteration 200 / 1000: loss 209.277646
iteration 300 / 1000: loss 79.226882
iteration 400 / 1000: loss 32.040598
iteration 500 / 1000: loss 15.247104
iteration 600 / 1000: loss 9.358830
iteration 700 / 1000: loss 6.757520
iteration 800 / 1000: loss 5.894799
iteration 900 / 1000: loss 5.962603
iteration 0 / 1000: loss 1745.710340
iteration 100 / 1000: loss 565.508734
iteration 200 / 1000: loss 186.084903
iteration 300 / 1000: loss 63.777366
iteration 800 / 1000: loss 6.187106
iteration 900 / 1000: loss 6.310308
iteration 0 / 1000: loss 3114.822663
iteration 100 / 1000: loss 417.448001
iteration 200 / 1000: loss 60.224435
iteration 300 / 1000: loss 13.267910
iteration 400 / 1000: loss 6.848183
iteration 500 / 1000: loss 6.508351
iteration 600 / 1000: loss 6.842337
iteration 700 / 1000: loss 6.476731
iteration 800 / 1000: loss 6.436558
iteration 900 / 1000: loss 6.706565
iteration 0 / 1000: loss 1541.456110
iteration 100 / 1000: loss 104.197655
iteration 200 / 1000: loss 12.334514
iteration 300 / 1000: loss 6.320392
iteration 400 / 1000: loss 5.519950
iteration 500 / 1000: loss 5.653650
iteration 600 / 1000: loss 5.865679
iteration 700 / 1000: loss 6.410740
iteration 800 / 1000: loss 5.459226
iteration 900 / 1000: loss 5.943483
iteration 0 / 1000: loss 1745.351504
iteration 100 / 1000: loss 86.203575
iteration 200 / 1000: loss 10.557921
iteration 300 / 1000: loss 6.111298
iteration 400 / 1000: loss 5.593837
iteration 500 / 1000: loss 6.015731
iteration 600 / 1000: loss 6.153673
iteration 700 / 1000: loss 5.807823
iteration 800 / 1000: loss 5.577925
iteration 900 / 1000: loss 5.437740
iteration 0 / 1000: loss 1861.981612
iteration 100 / 1000: loss 75.144166
iteration 200 / 1000: loss 8.922430
iteration 300 / 1000: loss 5.939499
iteration 400 / 1000: loss 6.004034
iteration 500 / 1000: loss 6.134184
iteration 600 / 1000: loss 6.343185
iteration 700 / 1000: loss 6.396444
iteration 800 / 1000: loss 6.556097
iteration 900 / 1000: loss 6.359304
iteration 0 / 1000: loss 3085.731047
iteration 100 / 1000: loss 19.132999
iteration 200 / 1000: loss 6.206095
iteration 300 / 1000: loss 5.881757
iteration 400 / 1000: loss 5.993762
iteration 500 / 1000: loss 6.703859
iteration 600 / 1000: loss 6.312134
iteration 700 / 1000: loss 6.263507
iteration 800 / 1000: loss 6.745562
iteration 900 / 1000: loss 6.244567
iteration 0 / 1000: loss 1561.022675
iteration 100 / 1000: loss 15.843969
iteration 200 / 1000: loss 6.439715
iteration 300 / 1000: loss 6.497490
iteration 400 / 1000: loss 5.774593
iteration 500 / 1000: loss 6.380114
iteration 600 / 1000: loss 6.657306
iteration 700 / 1000: loss 5.749389
iteration 800 / 1000: loss 6.261384
iteration 900 / 1000: loss 6.359610
iteration 0 / 1000: loss 1734.351602
iteration 100 / 1000: loss 12.119828
iteration 200 / 1000: loss 6.488434
iteration 300 / 1000: loss 6.113436
iteration 400 / 1000: loss 6.541479
iteration 500 / 1000: loss 7.073608
iteration 600 / 1000: loss 6.144135
iteration 700 / 1000: loss 6.305620
iteration 800 / 1000: loss 6.538148
iteration 900 / 1000: loss 6.077874
iteration 0 / 1000: loss 1880.962253
iteration 100 / 1000: loss 9.541188
iteration 200 / 1000: loss 6.615969
iteration 300 / 1000: loss 6.662919
iteration 400 / 1000: loss 5.861398
iteration 500 / 1000: loss 6.646663
iteration 600 / 1000: loss 6.427566
iteration 700 / 1000: loss 6.381281
iteration 800 / 1000: loss 5.958009
iteration 900 / 1000: loss 6.284874
iteration 0 / 1000: loss 3093.125754
iteration 100 / 1000: loss 6.549151
iteration 200 / 1000: loss 6.833877
iteration 300 / 1000: loss 6.334762
iteration 400 / 1000: loss 6.654829
iteration 500 / 1000: loss 6.509211
iteration 600 / 1000: loss 6.684489
iteration 700 / 1000: loss 6.720152
iteration 800 / 1000: loss 6.827805
iteration 900 / 1000: loss 7.025566
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.361980 val accuracy: 0.377000
lr 1.000000e-07 reg 2.800000e+04 train accuracy: 0.372653 val accuracy: 0.387000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.370082 val accuracy: 0.380000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.364612 val accuracy: 0.377000
lr 2.700000e-07 reg 2.500000e+04 train accuracy: 0.351265 val accuracy: 0.364000
lr 2.700000e-07 reg 2.800000e+04 train accuracy: 0.354755 val accuracy: 0.344000
lr 2.700000e-07 reg 3.000000e+04 train accuracy: 0.361061 val accuracy: 0.378000
lr 2.700000e-07 reg 5.000000e+04 train accuracy: 0.341469 val accuracy: 0.355000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.330755 val accuracy: 0.346000
lr 5.000000e-07 reg 2.800000e+04 train accuracy: 0.310959 val accuracy: 0.343000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.343020 val accuracy: 0.344000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.322347 val accuracy: 0.331000
best validation accuracy achieved during cross-validation: 0.387000
```

[73]:
```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()
```



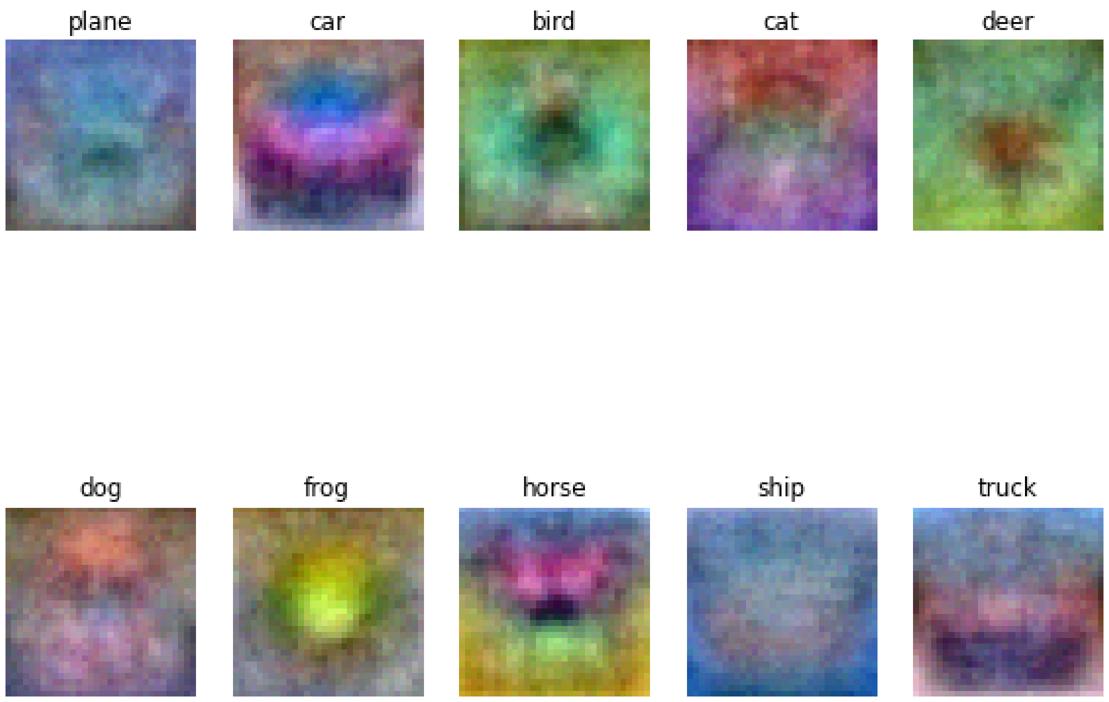


[74]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.366000
```

[75]:
```
# 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']
```

```python
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 the way they do.

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.