svm

September 29, 2021

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]:
```
# Setup Colab connection with Drive
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

# Define working directory
FOLDERNAME = 'Coursework/Fall 2021/682/Assignments/assignment1'
assert FOLDERNAME is not None, "[!] Enter the foldername."

import sys
sys.path.append(f'/content/gdrive/My Drive/{FOLDERNAME}')
```

```
Mounted at /content/gdrive
```

[2]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs682.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
```

[3]:
```
#Switch to working directory
%cd gdrive/My\ Drive/$FOLDERNAME
```

```
/content/gdrive/My Drive/Coursework/Fall 2021/682/Assignments/assignment1
```

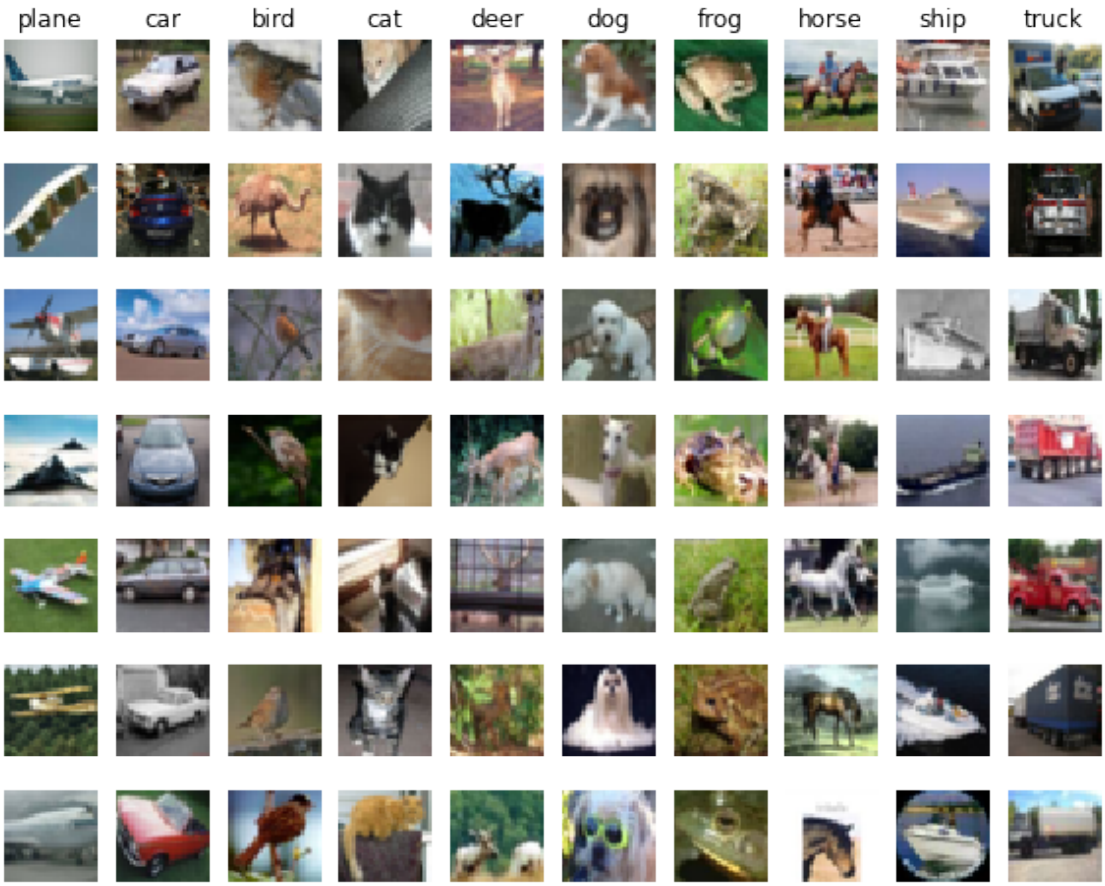
1.1 CIFAR-10 Data Loading and Preprocessing

[4]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/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,)
```

[5]:
```
# 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()
```



[6]:
```
# 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('Dev data shape: ', X_dev.shape)
print('Dev labels shape: ', y_dev.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,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

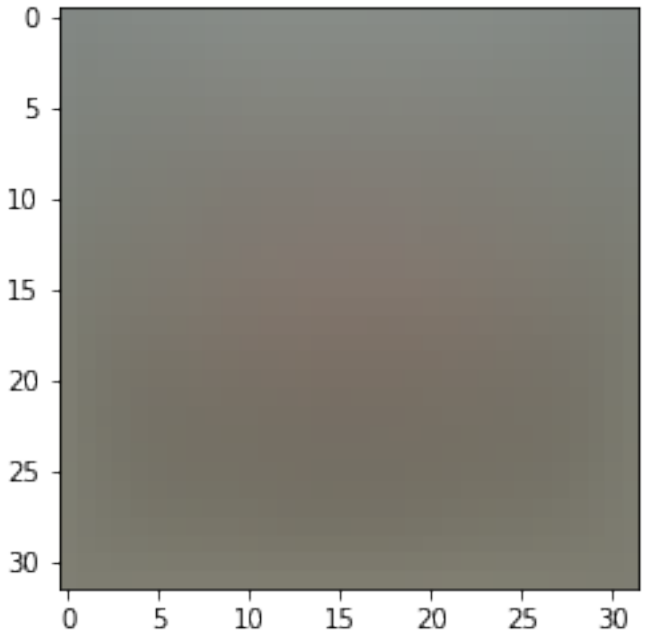
[7]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
```

```
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)
```

[8]:
```
# 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()
```

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



[9]:
```
# 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
```

[10]:
```
# 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)
```

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

1.2 SVM Classifier

Your code for this section will all be written inside **cs682/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.

[11]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs682.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: 8.809595
```

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:

[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 cs682.gradient_check import grad_check_sparse
print("Grad check")
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?
print("Grad check with Regularization")
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)
```

```
Grad check
numerical: -4.443697 analytic: -4.443697, relative error: 1.046154e-10
numerical: -47.410080 analytic: -47.410080, relative error: 4.717211e-13
numerical: 5.092056 analytic: 5.092056, relative error: 5.457824e-12
numerical: -21.319120 analytic: -21.319120, relative error: 2.776047e-12
numerical: -19.609801 analytic: -19.609801, relative error: 1.027689e-12
numerical: 15.476732 analytic: 15.476732, relative error: 1.398746e-11
numerical: 20.105124 analytic: 20.105124, relative error: 1.190624e-11
numerical: -8.229753 analytic: -8.229753, relative error: 5.555609e-11
numerical: 10.968917 analytic: 10.968917, relative error: 6.229928e-12
numerical: -8.471459 analytic: -8.471459, relative error: 3.769338e-11
Grad check with Regularization
numerical: -21.259623 analytic: -21.259623, relative error: 1.234623e-11
numerical: 28.185310 analytic: 28.185310, relative error: 6.668276e-12
numerical: -10.550279 analytic: -10.550279, relative error: 2.010631e-11
numerical: -2.744419 analytic: -2.744419, relative error: 1.813920e-10
numerical: -5.135584 analytic: -5.135584, relative error: 2.720993e-11
numerical: -4.570693 analytic: -4.570693, relative error: 1.048601e-11
numerical: -10.221300 analytic: -10.221300, relative error: 3.658213e-11
numerical: 0.337480 analytic: 0.337480, relative error: 1.626410e-09
numerical: 0.713218 analytic: 0.713218, relative error: 1.272639e-11
numerical: 13.958269 analytic: 13.958269, relative error: 4.373485e-12
```

1.2.1 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 changing the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: Yes it is possible that gradcheck may not match exactly. The SVM function being used is not differentiable on all points and at points where linearity is broken (i.e. the function is not differentiable), gradients won't be able to be calculated and discrepancies would

be created. This is more pronounced at points near origin (0,0) as the SVM function is of form max(0, x+m). If x=-0.01 and m=1, the function would have a kink and grad won't be calculated. A good way to deal with this is to properly set the margin parameter, or sample fewer data points while calculating the gradient.

[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 cs682.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: 8.809595e+00 computed in 0.116050s
Vectorized loss: 8.809595e+00 computed in 0.013691s
difference: 0.000000
```

[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 0.125974s
Vectorized loss and gradient: computed in 0.010834s
difference: 0.000000
```

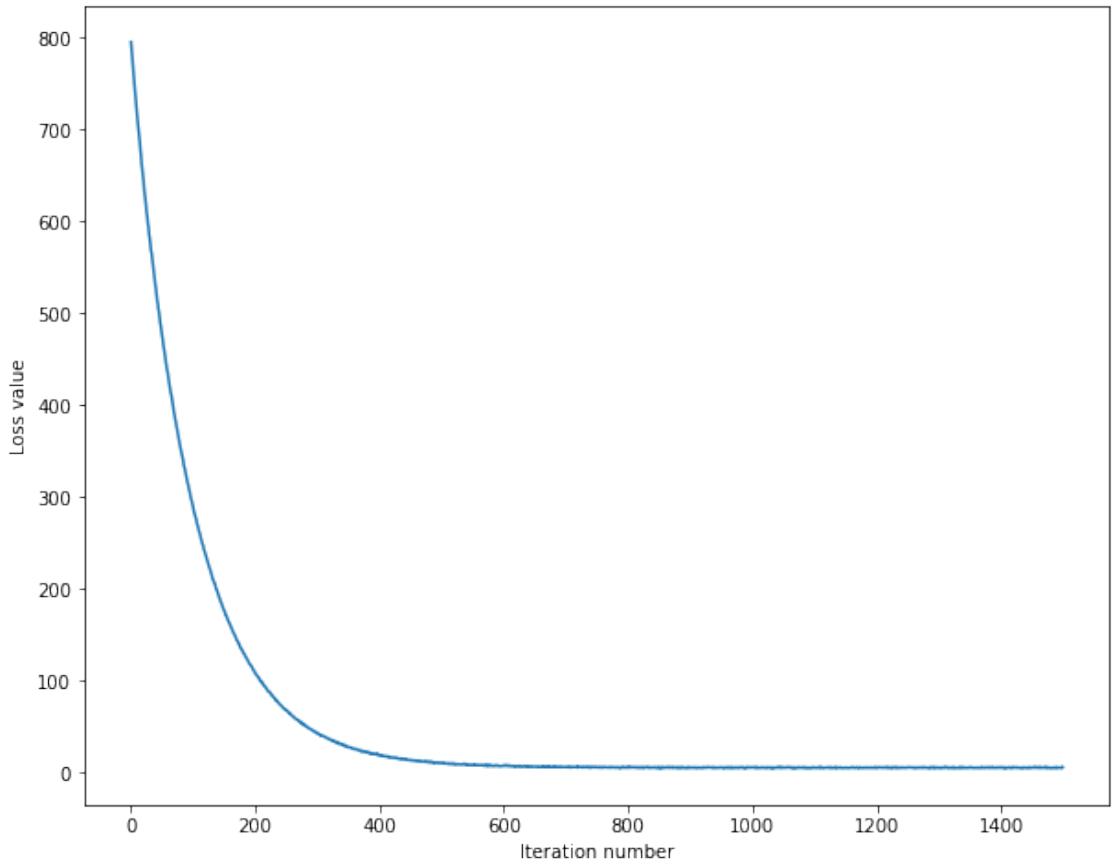
1.2.2 Stochastic Gradient Descent

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.

[15]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs682.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 795.315212
iteration 100 / 1500: loss 288.781899
iteration 200 / 1500: loss 107.602495
iteration 300 / 1500: loss 42.827045
iteration 400 / 1500: loss 19.035751
iteration 500 / 1500: loss 10.401494
iteration 600 / 1500: loss 7.468742
iteration 700 / 1500: loss 5.728218
iteration 800 / 1500: loss 6.245056
iteration 900 / 1500: loss 5.337141
iteration 1000 / 1500: loss 5.394541
iteration 1100 / 1500: loss 5.275675
iteration 1200 / 1500: loss 4.759179
iteration 1300 / 1500: loss 5.278764
iteration 1400 / 1500: loss 5.076657
That took 7.638534s
```

[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)
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: 0.371429
validation accuracy: 0.384000
```

[18]:
```
# 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 0.4 on the validation set.
learning_rates = [1e-7,9e-6, 7e-6, 5e-6, 3e-6, 1e-6, 9e-5, 7e-5, 5e-5]
regularization_strengths = [2.5e4, 3e4, 3.5e4, 4e4, 4.5e4, 5e4]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
```

```
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation␣
 ↪rate.

################################################################################
# TODO:                                                                        ␣
 ↪#
# Write code that chooses the best hyperparameters by tuning on the validation␣
 ↪#
# set. For each combination of hyperparameters, train a linear SVM on the      ␣
 ↪#
# training set, compute its accuracy on the training and validation sets, and ␣
 ↪#
# store these numbers in the results dictionary. In addition, store the best   ␣
 ↪#
# validation accuracy in best_val and the LinearSVM object that achieves this ␣
 ↪#
# accuracy in best_svm.                                                        ␣
 ↪#
#                                                                              ␣
 ↪#
# Hint: You should use a small value for num_iters as you develop your         ␣
 ↪#
# validation code so that the SVMs don't take much time to train; once you are␣
 ↪#
# confident that your validation code works, you should rerun the validation   ␣
 ↪#
# code with a larger value for num_iters.                                      ␣
 ↪#
################################################################################
# Your code
with np.errstate(all='ignore'):
  for lr in learning_rates:
    for r in regularization_strengths:
      svm = LinearSVM()
      loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=r,␣
 ↪num_iters=1500, verbose=False)
      y_train_pred = svm.predict(X_train)
      train_acc = np.mean(y_train == y_train_pred)
      y_val_pred = svm.predict(X_val)
      val_acc = np.mean(y_val == y_val_pred)
      results[(lr,r)] = (train_acc, val_acc)
      if val_acc > best_val:
        best_val = val_acc
```

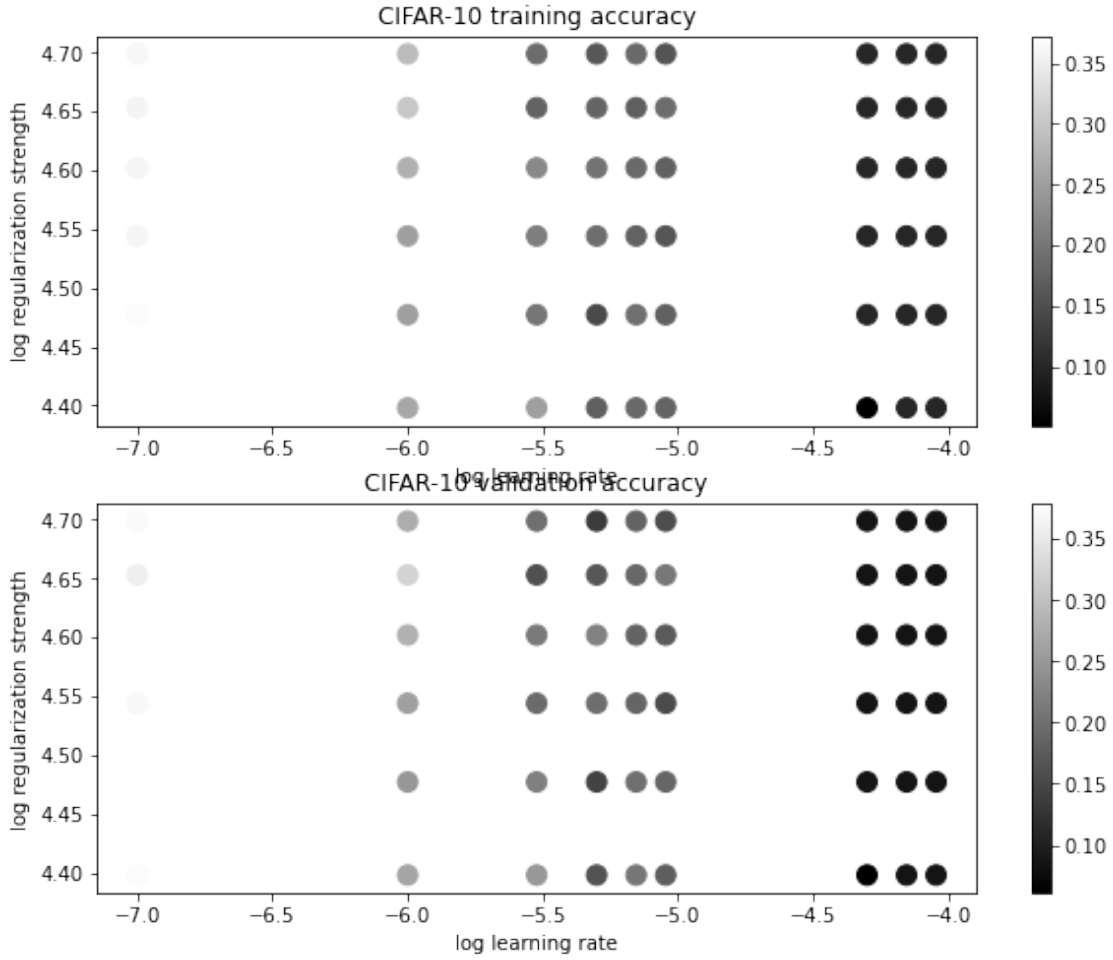
```
        best_svm = svm

################################################################################
#                               END OF YOUR CODE                               #
################################################################################

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

print('best validation accuracy achieved during cross-validation: %f' %␣
↪best_val)
#lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.369184 val accuracy: 0.
↪395000
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.371980 val accuracy: 0.373000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.366796 val accuracy: 0.379000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.358347 val accuracy: 0.370000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.359286 val accuracy: 0.378000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.357184 val accuracy: 0.358000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.360939 val accuracy: 0.372000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.260959 val accuracy: 0.268000
lr 1.000000e-06 reg 3.000000e+04 train accuracy: 0.251735 val accuracy: 0.249000
lr 1.000000e-06 reg 3.500000e+04 train accuracy: 0.251816 val accuracy: 0.260000
lr 1.000000e-06 reg 4.000000e+04 train accuracy: 0.272122 val accuracy: 0.281000
lr 1.000000e-06 reg 4.500000e+04 train accuracy: 0.301245 val accuracy: 0.324000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.288061 val accuracy: 0.276000
lr 3.000000e-06 reg 2.500000e+04 train accuracy: 0.251714 val accuracy: 0.251000
lr 3.000000e-06 reg 3.000000e+04 train accuracy: 0.200245 val accuracy: 0.221000
lr 3.000000e-06 reg 3.500000e+04 train accuracy: 0.210000 val accuracy: 0.194000
lr 3.000000e-06 reg 4.000000e+04 train accuracy: 0.223306 val accuracy: 0.213000
lr 3.000000e-06 reg 4.500000e+04 train accuracy: 0.175959 val accuracy: 0.162000
lr 3.000000e-06 reg 5.000000e+04 train accuracy: 0.187531 val accuracy: 0.200000
lr 5.000000e-06 reg 2.500000e+04 train accuracy: 0.171776 val accuracy: 0.164000
lr 5.000000e-06 reg 3.000000e+04 train accuracy: 0.145327 val accuracy: 0.145000
lr 5.000000e-06 reg 3.500000e+04 train accuracy: 0.190265 val accuracy: 0.198000
lr 5.000000e-06 reg 4.000000e+04 train accuracy: 0.195878 val accuracy: 0.223000
lr 5.000000e-06 reg 4.500000e+04 train accuracy: 0.178327 val accuracy: 0.170000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.163163 val accuracy: 0.135000
lr 7.000000e-06 reg 2.500000e+04 train accuracy: 0.184327 val accuracy: 0.210000
lr 7.000000e-06 reg 3.000000e+04 train accuracy: 0.192837 val accuracy: 0.201000
lr 7.000000e-06 reg 3.500000e+04 train accuracy: 0.174531 val accuracy: 0.187000
lr 7.000000e-06 reg 4.000000e+04 train accuracy: 0.185776 val accuracy: 0.184000
lr 7.000000e-06 reg 4.500000e+04 train accuracy: 0.171571 val accuracy: 0.190000
```

```
lr 7.000000e-06 reg 5.000000e+04 train accuracy: 0.184653 val accuracy: 0.185000
lr 9.000000e-06 reg 2.500000e+04 train accuracy: 0.178041 val accuracy: 0.178000
lr 9.000000e-06 reg 3.000000e+04 train accuracy: 0.173347 val accuracy: 0.188000
lr 9.000000e-06 reg 3.500000e+04 train accuracy: 0.158714 val accuracy: 0.155000
lr 9.000000e-06 reg 4.000000e+04 train accuracy: 0.173347 val accuracy: 0.174000
lr 9.000000e-06 reg 4.500000e+04 train accuracy: 0.187429 val accuracy: 0.211000
lr 9.000000e-06 reg 5.000000e+04 train accuracy: 0.158265 val accuracy: 0.157000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.052061 val accuracy: 0.061000
lr 5.000000e-05 reg 3.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 3.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 4.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 4.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 3.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 3.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 4.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 4.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 9.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 9.000000e-05 reg 3.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 9.000000e-05 reg 3.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 9.000000e-05 reg 4.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 9.000000e-05 reg 4.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 9.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.379000
```

[19]:
```python
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
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)
```

```
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

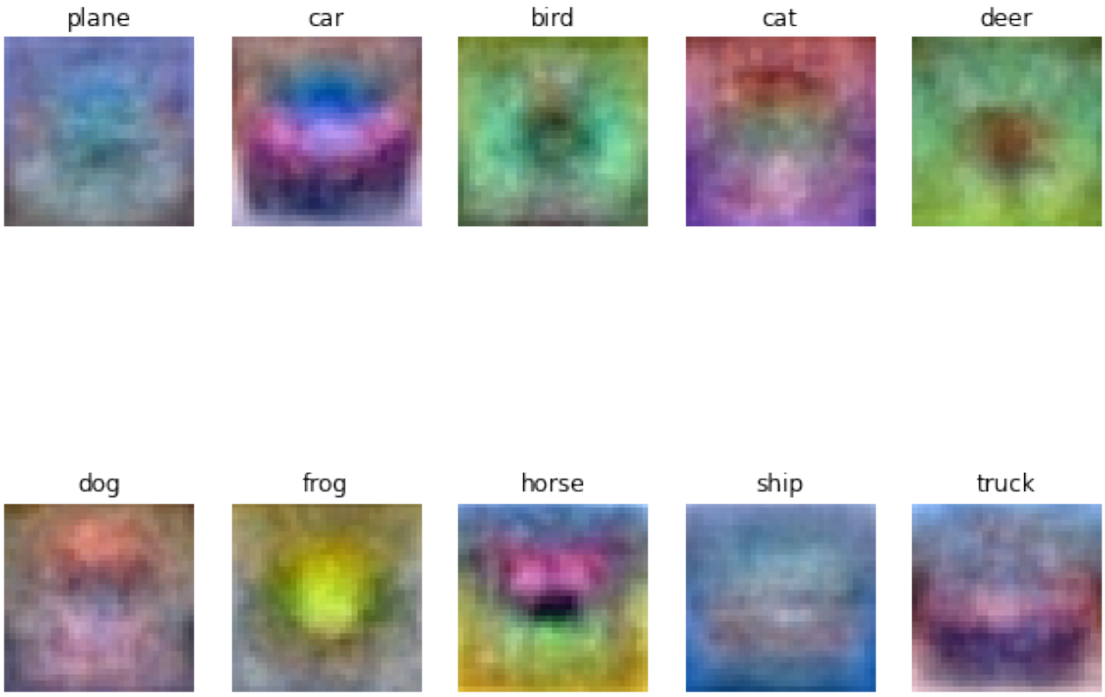


[20]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.360000
```

[21]:
```
# 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])
```



1.2.3 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 visualized SVM weights look like hazy, abstract or pixelated versions of the entities they represent. For instance, the outline of a bonnet can be made out for the car weight representation, while a two-headed horse can be made out from the horse weight representation. The weights can be thought of as a template for the image and hence they look like a rough idea of the class.