# svm

December 5, 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

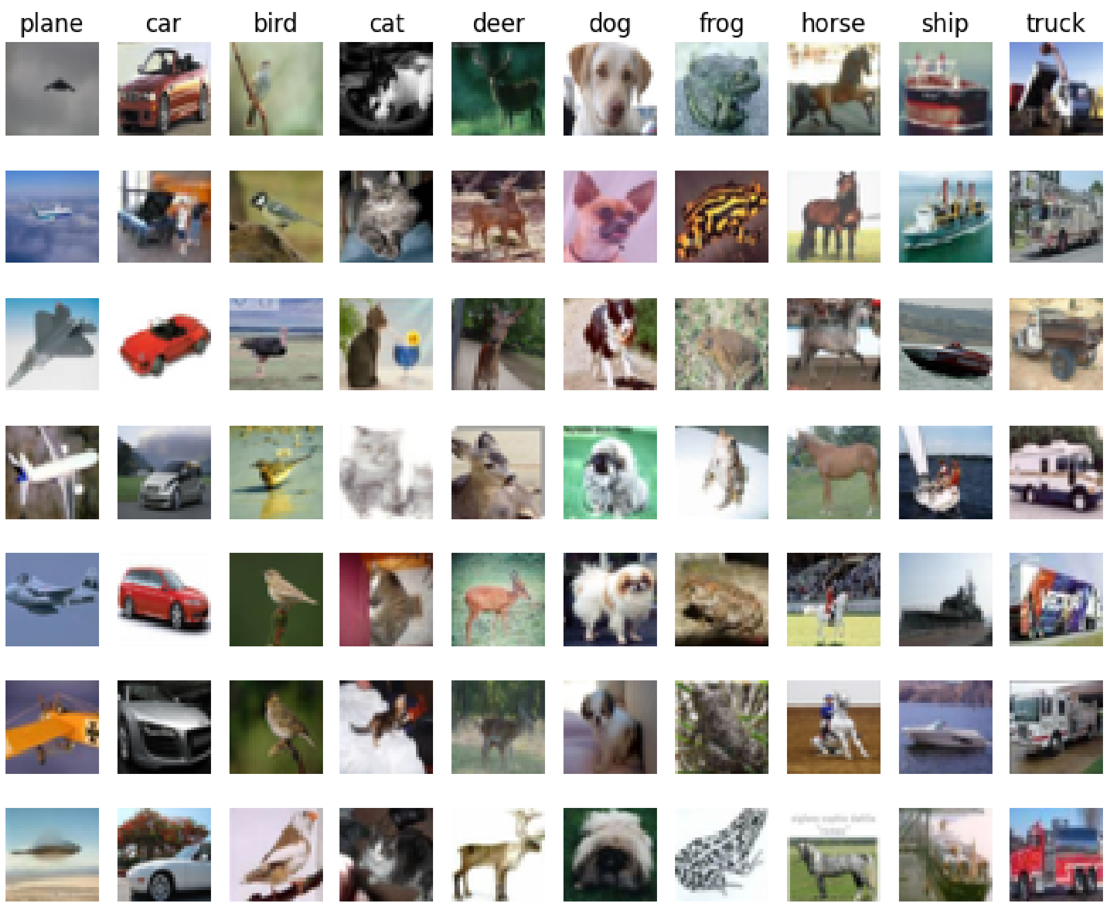
[ ]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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

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

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



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

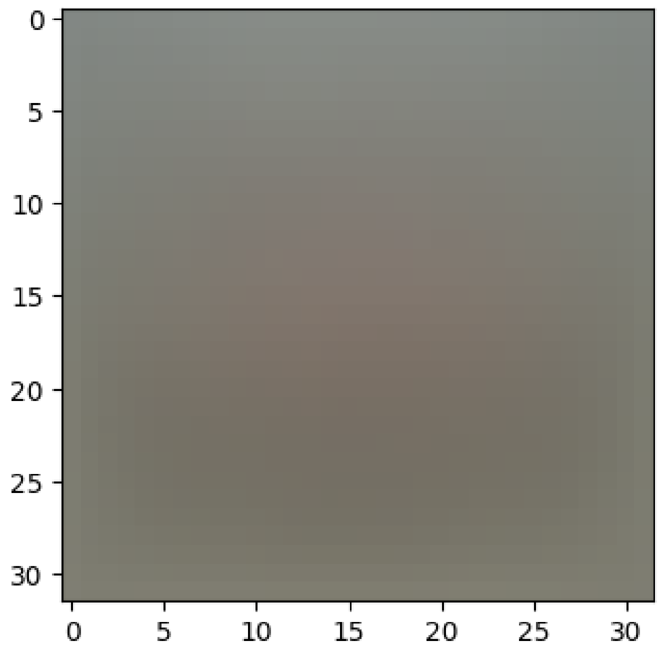
[ ]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

[ ]:
```
# 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.

```
[ ]: # 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

     # we should download the 'past' module by 'pip install future' ????!!!!

     loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
     print('loss: %f' % (loss, ))
```

```
loss: 9.191689
```

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:

```
[ ]: # 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?

     print("=============== with regularization turned on ===============")
     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: 9.507262 analytic: 9.507262, relative error: 4.365145e-11
numerical: -10.402340 analytic: -10.402340, relative error: 2.542283e-11
numerical: -6.788093 analytic: -6.788093, relative error: 9.900149e-12
numerical: 26.288257 analytic: 26.288257, relative error: 8.893590e-12
numerical: 6.568668 analytic: 6.568668, relative error: 3.922110e-11
numerical: -2.864591 analytic: -2.864591, relative error: 2.109229e-10
numerical: -5.385446 analytic: -5.385446, relative error: 4.948583e-11
numerical: -4.102524 analytic: -4.102524, relative error: 1.472608e-10
numerical: -22.175930 analytic: -22.175930, relative error: 9.105219e-12
numerical: 23.711867 analytic: 23.711867, relative error: 1.137835e-11
=============== with regularization turned on ===============
numerical: 21.596111 analytic: 21.596111, relative error: 4.447450e-13
numerical: 14.912787 analytic: 14.912787, relative error: 2.424971e-11
numerical: 4.573528 analytic: 4.573528, relative error: 2.641263e-11
numerical: -6.542642 analytic: -6.542642, relative error: 4.862198e-11
numerical: 2.132395 analytic: 2.132395, relative error: 1.584846e-12
numerical: 2.903656 analytic: 2.903656, relative error: 1.134656e-10
numerical: 9.541348 analytic: 9.541348, relative error: 2.156265e-11
numerical: 10.123972 analytic: 10.123972, relative error: 1.606656e-11
numerical: 9.336602 analytic: 9.336602, relative error: 4.154354e-11
numerical: 4.055225 analytic: 4.055225, relative error: 7.665387e-11
```

**Inline Question 1**

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

$Your Answer:$

1. It is caused by the usage of taking max, which caused the loss function not differentiable at the point whose the margin is close to 0. As the delta is set to be 1, so for these points, $score[j] \to 0$, and $score[y[i]] \to 1$, which is close to the correct classification.

2. No, it is not reason for concern. Only a few effect will be caused by the indifferentiable point. Which occur in low probability. For most situation, the loss function is differentiable. And the where margin is equal to 0 are the points we want to reach, so it will not a reason for concern.

3. For example, in one dimension, consider the $ReLU$ function $f(x) = \max(0, x)$.
   We $x$ is close to 0, for example, we take $x = \dfrac{\epsilon}{2}$, where $\epsilon \to 0$.

   - For numerical gradient, we have grad_numerical $= \dfrac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} = \dfrac{\max(0, \frac{\epsilon}{2}+\epsilon) - \max(0, \dfrac{\epsilon}{2} - \epsilon)}{2\epsilon} = \dfrac{\frac{3}{2}\epsilon}{2\epsilon} = \dfrac{3}{4}$

- For analytical gradient, since when $x = \epsilon > 0$, the function $f(x)$ is differentiable, and $f'(x) = 1$,
  so we have grad_analytical $= f'(\epsilon) = 1$.

So the when $x \to 0$, for example in above, the gradient check will fail.

4. The frequency of the failure of graient check happening will increase, as the margin gets close to 0.

[ ]:
```
# 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: 9.191689e+00 computed in 0.112915s
Vectorized loss: 9.191689e+00 computed in 0.003833s
difference: -0.000000
```

[ ]:
```
# 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 0.111945s
Vectorized loss and gradient: computed in 0.002966s
difference: 0.000000
```

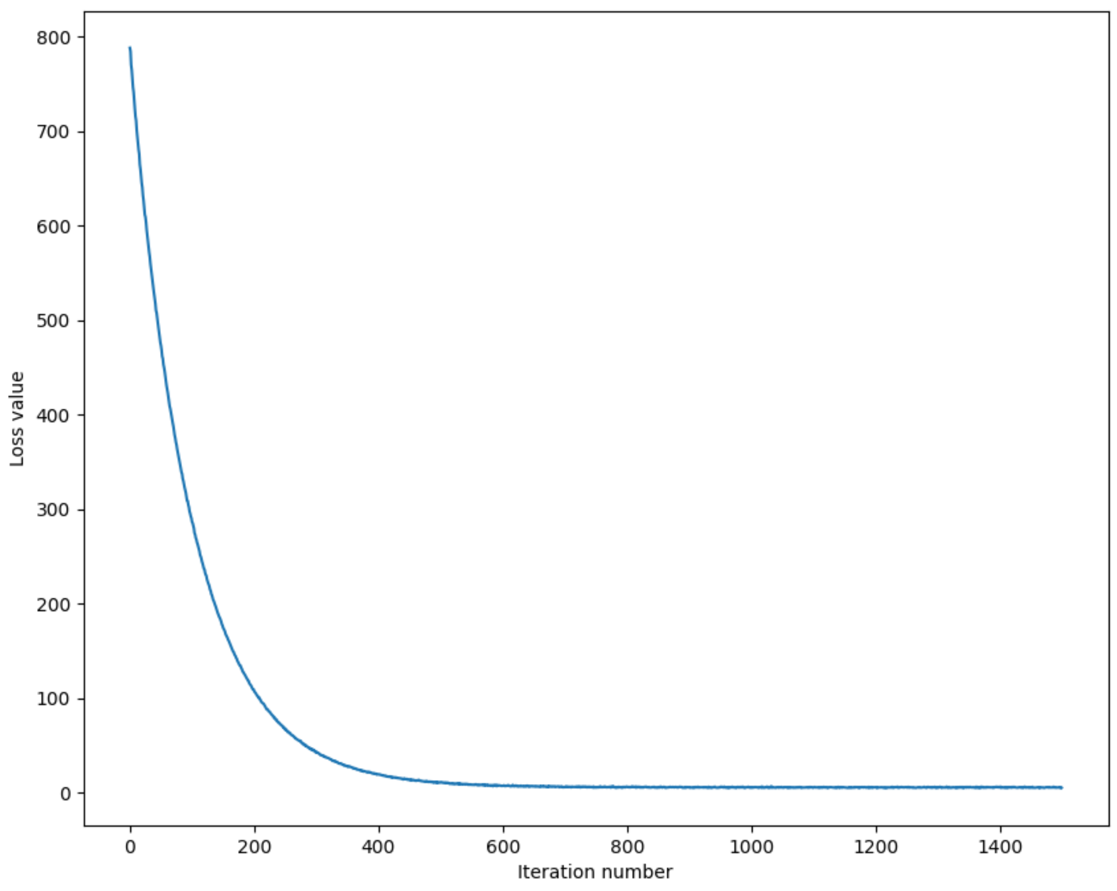
### 1.2.1 Stochastic Gradient Descent

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

[ ]:
```
# In 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 787.957425
iteration 100 / 1500: loss 286.196665
iteration 200 / 1500: loss 107.259840
iteration 300 / 1500: loss 43.037508
iteration 400 / 1500: loss 19.082985
iteration 500 / 1500: loss 10.890695
iteration 600 / 1500: loss 6.967051
iteration 700 / 1500: loss 5.797226
iteration 800 / 1500: loss 5.950527
iteration 900 / 1500: loss 5.461388
iteration 1000 / 1500: loss 5.619386
iteration 1100 / 1500: loss 5.147947
iteration 1200 / 1500: loss 5.502244
iteration 1300 / 1500: loss 4.944740
iteration 1400 / 1500: loss 5.600263
That took 3.034621s
```

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



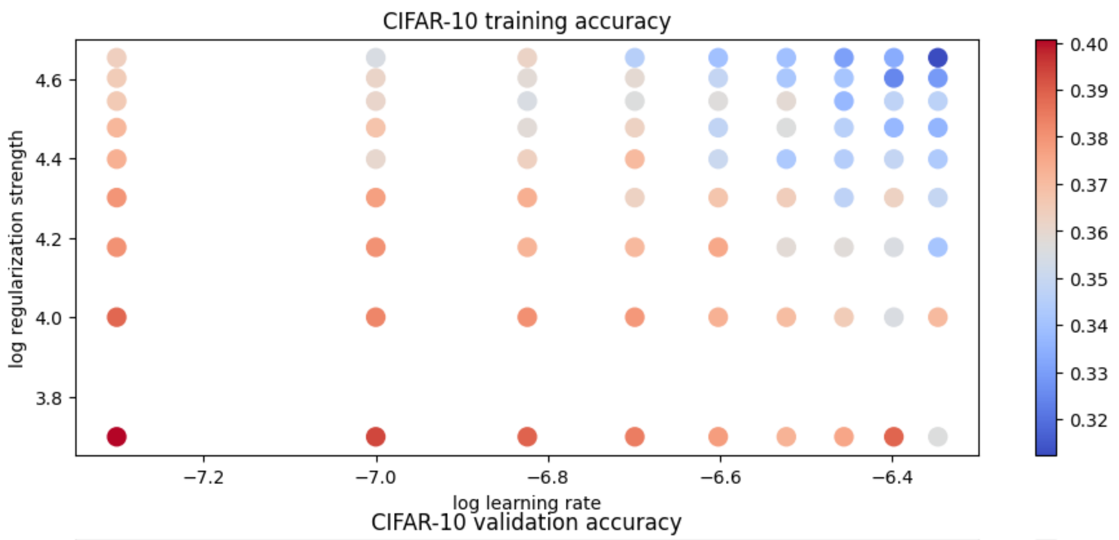
```
[ ]: # Write the LinearSVM.predict function and evaluate the performance on both the
     # training and validation set
     y_train_pred = svm.predict(X_train)
     print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
     y_val_pred = svm.predict(X_val)
     print('validation accuracy: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

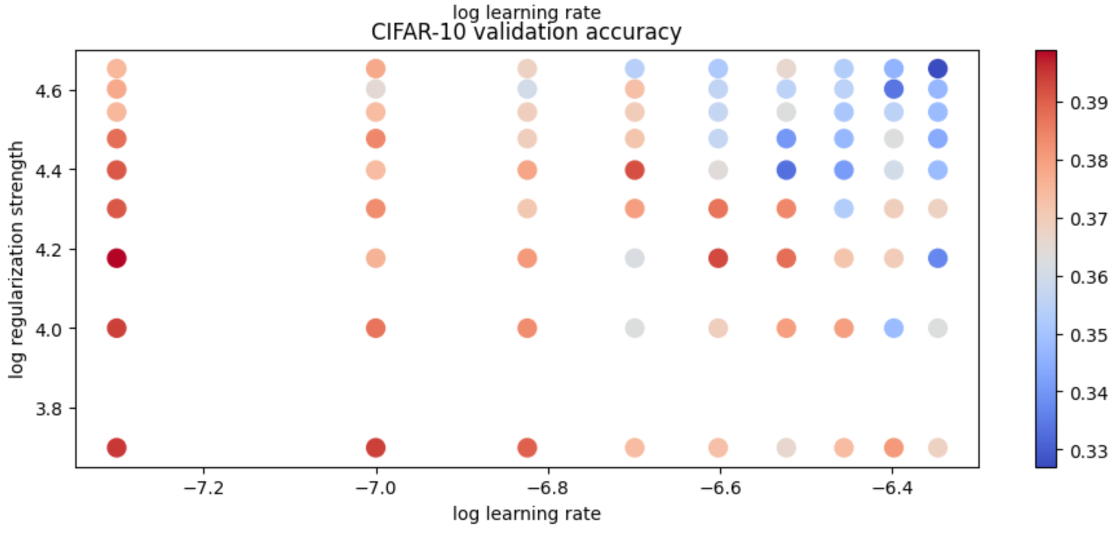
```
training accuracy: 0.374633
validation accuracy: 0.376000
```

```
[ ]: # 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, 5e-5]
     # regularization_strengths = [2.5e4, 5e4]

     delta_lr = 5e-8
     delta_reg = 5e3
     learning_rates = [lr * delta_lr for lr in range(1, 10)]
     regularization_strengths = [reg * delta_reg for reg in range(1, 10)]


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

     # pass

     iter = 5000
     for lr in learning_rates:
         for reg in regularization_strengths:
             svm = LinearSVM()
             loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=reg,␣
      ↪num_iters=iter)

             y_train_pred = svm.predict(X_train)
             y_val_pred = svm.predict(X_val)

             train_accuracy = np.mean(y_train == y_train_pred)
             val_accuracy = np.mean(y_val == y_val_pred)

             results[(lr, reg)] = (train_accuracy, val_accuracy)
             if val_accuracy > best_val:
                 best_val = val_accuracy
                 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)
```

```
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.400837 val accuracy: 0.395000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.388633 val accuracy: 0.394000
lr 5.000000e-08 reg 1.500000e+04 train accuracy: 0.380204 val accuracy: 0.399000
lr 5.000000e-08 reg 2.000000e+04 train accuracy: 0.379510 val accuracy: 0.391000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.373204 val accuracy: 0.391000
lr 5.000000e-08 reg 3.000000e+04 train accuracy: 0.371735 val accuracy: 0.388000
lr 5.000000e-08 reg 3.500000e+04 train accuracy: 0.365571 val accuracy: 0.375000
lr 5.000000e-08 reg 4.000000e+04 train accuracy: 0.365163 val accuracy: 0.378000
lr 5.000000e-08 reg 4.500000e+04 train accuracy: 0.363776 val accuracy: 0.375000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.393673 val accuracy: 0.394000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.382633 val accuracy: 0.387000
lr 1.000000e-07 reg 1.500000e+04 train accuracy: 0.380122 val accuracy: 0.376000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.377041 val accuracy: 0.383000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.360204 val accuracy: 0.374000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.367898 val accuracy: 0.384000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.360694 val accuracy: 0.374000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.361388 val accuracy: 0.365000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.355041 val accuracy: 0.378000
lr 1.500000e-07 reg 5.000000e+03 train accuracy: 0.389122 val accuracy: 0.390000
lr 1.500000e-07 reg 1.000000e+04 train accuracy: 0.380735 val accuracy: 0.383000
lr 1.500000e-07 reg 1.500000e+04 train accuracy: 0.372327 val accuracy: 0.381000
lr 1.500000e-07 reg 2.000000e+04 train accuracy: 0.373776 val accuracy: 0.371000
lr 1.500000e-07 reg 2.500000e+04 train accuracy: 0.363327 val accuracy: 0.379000
lr 1.500000e-07 reg 3.000000e+04 train accuracy: 0.357959 val accuracy: 0.369000
lr 1.500000e-07 reg 3.500000e+04 train accuracy: 0.355082 val accuracy: 0.369000
lr 1.500000e-07 reg 4.000000e+04 train accuracy: 0.358347 val accuracy: 0.360000
lr 1.500000e-07 reg 4.500000e+04 train accuracy: 0.361755 val accuracy: 0.368000
lr 2.000000e-07 reg 5.000000e+03 train accuracy: 0.384490 val accuracy: 0.374000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.378694 val accuracy: 0.363000
lr 2.000000e-07 reg 1.500000e+04 train accuracy: 0.371000 val accuracy: 0.362000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.362449 val accuracy: 0.380000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.370694 val accuracy: 0.392000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.362490 val accuracy: 0.372000
lr 2.000000e-07 reg 3.500000e+04 train accuracy: 0.356265 val accuracy: 0.370000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.358980 val accuracy: 0.373000
lr 2.000000e-07 reg 4.500000e+04 train accuracy: 0.345347 val accuracy: 0.354000
lr 2.500000e-07 reg 5.000000e+03 train accuracy: 0.377796 val accuracy: 0.373000
lr 2.500000e-07 reg 1.000000e+04 train accuracy: 0.373082 val accuracy: 0.369000
lr 2.500000e-07 reg 1.500000e+04 train accuracy: 0.375531 val accuracy: 0.393000
lr 2.500000e-07 reg 2.000000e+04 train accuracy: 0.367449 val accuracy: 0.387000
lr 2.500000e-07 reg 2.500000e+04 train accuracy: 0.350612 val accuracy: 0.364000
lr 2.500000e-07 reg 3.000000e+04 train accuracy: 0.348061 val accuracy: 0.357000
lr 2.500000e-07 reg 3.500000e+04 train accuracy: 0.357102 val accuracy: 0.357000
lr 2.500000e-07 reg 4.000000e+04 train accuracy: 0.348735 val accuracy: 0.356000
lr 2.500000e-07 reg 4.500000e+04 train accuracy: 0.339592 val accuracy: 0.352000
lr 3.000000e-07 reg 5.000000e+03 train accuracy: 0.372653 val accuracy: 0.366000
lr 3.000000e-07 reg 1.000000e+04 train accuracy: 0.369837 val accuracy: 0.380000
lr 3.000000e-07 reg 1.500000e+04 train accuracy: 0.358449 val accuracy: 0.388000
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.364449 val accuracy: 0.384000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.342388 val accuracy: 0.333000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.356245 val accuracy: 0.340000
lr 3.000000e-07 reg 3.500000e+04 train accuracy: 0.358816 val accuracy: 0.363000
lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.341673 val accuracy: 0.355000
lr 3.000000e-07 reg 4.500000e+04 train accuracy: 0.339306 val accuracy: 0.366000
lr 3.500000e-07 reg 5.000000e+03 train accuracy: 0.375735 val accuracy: 0.374000
lr 3.500000e-07 reg 1.000000e+04 train accuracy: 0.364837 val accuracy: 0.380000
lr 3.500000e-07 reg 1.500000e+04 train accuracy: 0.357755 val accuracy: 0.372000
lr 3.500000e-07 reg 2.000000e+04 train accuracy: 0.346918 val accuracy: 0.353000
lr 3.500000e-07 reg 2.500000e+04 train accuracy: 0.344633 val accuracy: 0.341000
lr 3.500000e-07 reg 3.000000e+04 train accuracy: 0.345490 val accuracy: 0.347000
lr 3.500000e-07 reg 3.500000e+04 train accuracy: 0.336959 val accuracy: 0.351000
lr 3.500000e-07 reg 4.000000e+04 train accuracy: 0.340980 val accuracy: 0.355000
lr 3.500000e-07 reg 4.500000e+04 train accuracy: 0.330531 val accuracy: 0.353000
lr 4.000000e-07 reg 5.000000e+03 train accuracy: 0.388735 val accuracy: 0.381000
lr 4.000000e-07 reg 1.000000e+04 train accuracy: 0.355286 val accuracy: 0.348000
lr 4.000000e-07 reg 1.500000e+04 train accuracy: 0.355143 val accuracy: 0.370000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.362429 val accuracy: 0.369000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.348673 val accuracy: 0.360000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.336796 val accuracy: 0.363000
lr 4.000000e-07 reg 3.500000e+04 train accuracy: 0.347408 val accuracy: 0.355000
lr 4.000000e-07 reg 4.000000e+04 train accuracy: 0.324592 val accuracy: 0.334000
lr 4.000000e-07 reg 4.500000e+04 train accuracy: 0.333653 val accuracy: 0.346000
lr 4.500000e-07 reg 5.000000e+03 train accuracy: 0.356408 val accuracy: 0.368000
lr 4.500000e-07 reg 1.000000e+04 train accuracy: 0.370633 val accuracy: 0.363000
lr 4.500000e-07 reg 1.500000e+04 train accuracy: 0.340959 val accuracy: 0.337000
lr 4.500000e-07 reg 2.000000e+04 train accuracy: 0.349245 val accuracy: 0.368000
lr 4.500000e-07 reg 2.500000e+04 train accuracy: 0.343245 val accuracy: 0.348000
lr 4.500000e-07 reg 3.000000e+04 train accuracy: 0.336020 val accuracy: 0.344000
lr 4.500000e-07 reg 3.500000e+04 train accuracy: 0.346612 val accuracy: 0.348000
lr 4.500000e-07 reg 4.000000e+04 train accuracy: 0.329061 val accuracy: 0.347000
lr 4.500000e-07 reg 4.500000e+04 train accuracy: 0.312204 val accuracy: 0.327000
best validation accuracy achieved during cross-validation: 0.399000
```

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



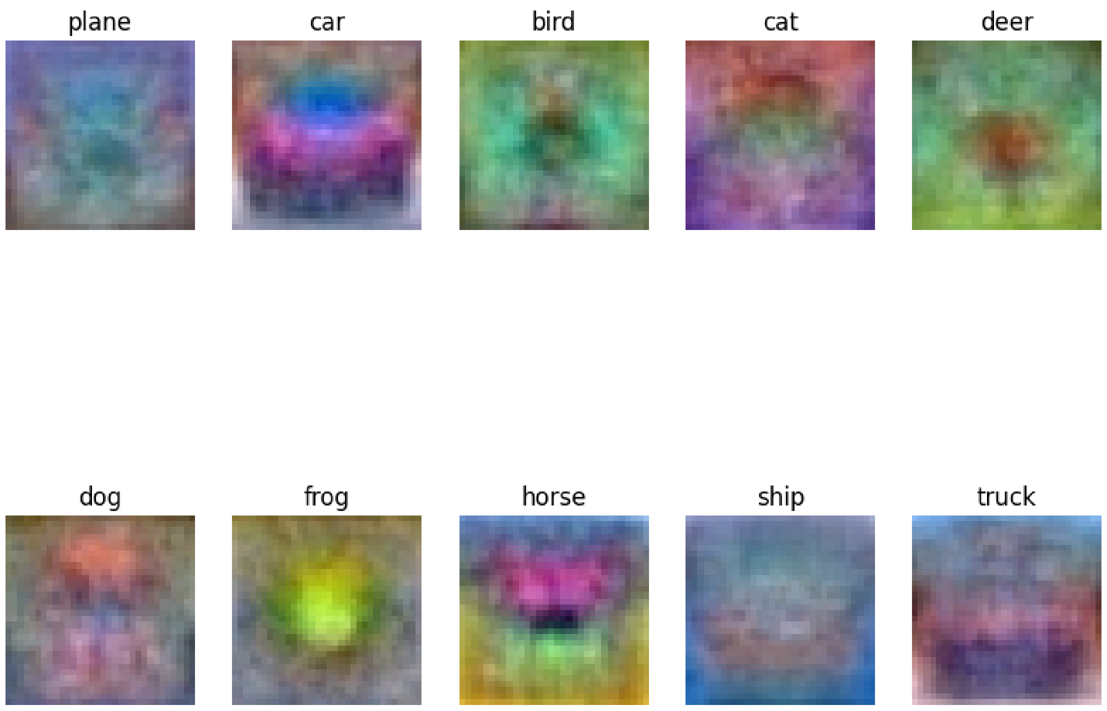


```
[ ]: # Evaluate the best svm on test set
     y_test_pred = best_svm.predict(X_test)
     test_accuracy = np.mean(y_test == y_test_pred)
     print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.383000
```

```
[ ]: # 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 they way that they do.

$\color{blue}{Your Answer:}$

The weight visualization of each class roughly shows the background color and shape of the object. But as it is trained on various images, the shape is not very clear, and may have some noise. The image seems blurry, but somehow we can still see the shape of the object, and the background color of the object.

For example, for the weights of horses, we could vaguely see that the weight visualization seems to have two heads, which is due to the fact that the horses in the dataset are facing different directions. The weight visualization of the frog shows the color of the frog image, the weight visualization of the ship shows the blue color of the sea, which means that there are more ships on the sea in the dataset……

And for the car, we could see that the weight visualization shows the shape of the car, and the color of the car is red, which means that there are more red cars in the dataset than other colors. Or the average color of the cars in the dataset is red.

So above all, although the weight visualization is not very clear, it still roughly shows the shape and color of the object, which is used to recognize the object. It learned the template of each class.