linear_classifier

July 24, 2022

1 EECS 498-007/598-005 Assignment 2-1: Linear Classifiers

Before we start, please put your name and UMID in following format Firstname LASTNAME, #00000000 // e.g.) Justin JOHNSON, #12345678

Your Answer:
Joseph CAMACHO, #XXXXXXXX

1.1 Install starter code

We will continue using the utility functions that we've used for Assignment 1: `coutils` package. Run this cell to download and install it.

```
[ ]: !pip install git+https://github.com/deepvision-class/starter-code
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting git+https://github.com/deepvision-class/starter-code
  Cloning https://github.com/deepvision-class/starter-code to /tmp/pip-req-
build-xwygbz6g
  Running command git clone -q https://github.com/deepvision-class/starter-code
/tmp/pip-req-build-xwygbz6g
Requirement already satisfied: pydrive in /usr/local/lib/python3.7/dist-packages
(from Colab-Utils==0.1.dev0) (1.3.1)
Requirement already satisfied: PyYAML>=3.0 in /usr/local/lib/python3.7/dist-
packages (from pydrive->Colab-Utils==0.1.dev0) (3.13)
Requirement already satisfied: google-api-python-client>=1.2 in
/usr/local/lib/python3.7/dist-packages (from pydrive->Colab-Utils==0.1.dev0)
(1.12.11)
Requirement already satisfied: oauth2client>=4.0.0 in
/usr/local/lib/python3.7/dist-packages (from pydrive->Colab-Utils==0.1.dev0)
(4.1.3)
Requirement already satisfied: google-auth<3dev,>=1.16.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (1.35.0)
Requirement already satisfied: httplib2<1dev,>=0.15.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (0.17.4)
Requirement already satisfied: google-api-core<3dev,>=1.21.0 in
```

```
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (1.31.6)
Requirement already satisfied: google-auth-httplib2>=0.0.3 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (0.0.4)
Requirement already satisfied: six<2dev,>=1.13.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (1.15.0)
Requirement already satisfied: uritemplate<4dev,>=3.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (3.0.1)
Requirement already satisfied: packaging>=14.3 in /usr/local/lib/python3.7/dist-
packages (from google-api-core<3dev,>=1.21.0->google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (21.3)
Requirement already satisfied: googleapis-common-protos<2.0dev,>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (1.56.4)
Requirement already satisfied: setuptools>=40.3.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (57.4.0)
Requirement already satisfied: protobuf<4.0.0dev,>=3.12.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (3.17.3)
Requirement already satisfied: pytz in /usr/local/lib/python3.7/dist-packages
(from google-api-core<3dev,>=1.21.0->google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (2022.1)
Requirement already satisfied: requests<3.0.0dev,>=2.18.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (2.23.0)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-
packages (from google-auth<3dev,>=1.16.0->google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (4.8)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-
api-python-client>=1.2->pydrive->Colab-Utils==0.1.dev0) (0.2.8)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-
api-python-client>=1.2->pydrive->Colab-Utils==0.1.dev0) (4.2.4)
Requirement already satisfied: pyasn1>=0.1.7 in /usr/local/lib/python3.7/dist-
packages (from oauth2client>=4.0.0->pydrive->Colab-Utils==0.1.dev0) (0.4.8)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging>=14.3->google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (3.0.9)
```

```
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-
api-core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests<3.0.0dev,>=2.18.0->google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-
api-core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (2022.6.15)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-
api-core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (3.0.4)
```

1.2 Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

[ ]:
```python
from __future__ import print_function
from __future__ import division

import torch
import coutils
import random
import time
import math
import matplotlib.pyplot as plt
from torchvision.utils import make_grid

%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'
```

Starting in this assignment, we will use the GPU to accelerate our computation. Run this cell to make sure you are using a GPU.

[ ]:
```python
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

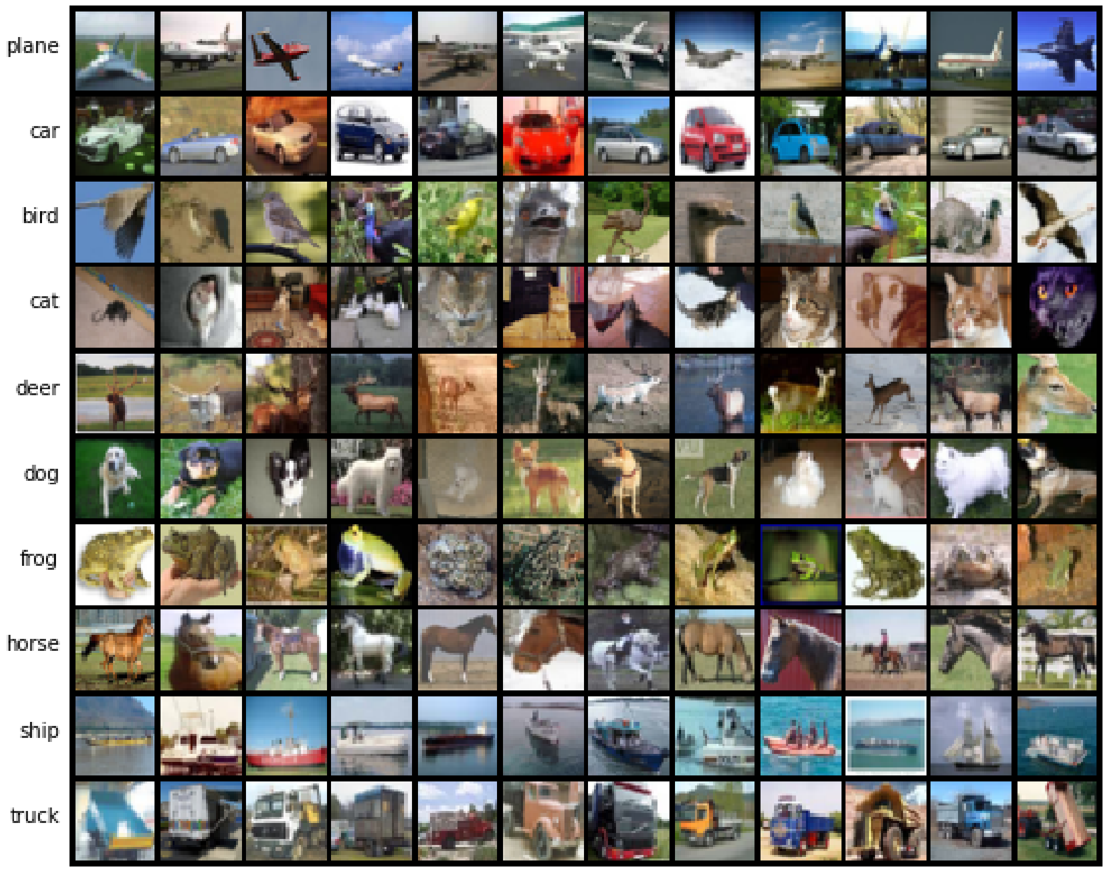
Now, we will load CIFAR10 dataset, with normalization.

In this notebook we will use the **bias trick**: By adding an extra constant feature of ones to each image, we avoid the need to keep track of a bias vector; the bias will be encoded as the part of the weight matrix that interacts with the constant ones in the input.

In the `two_layer_net.ipynb` notebook that follows this one, we will not use the bias trick.

```
[ ]: def get_CIFAR10_data(validation_ratio = 0.02):
       """
       Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
       it for the linear classifier. These are the same steps as we used for the
       SVM, but condensed to a single function.
       """
       X_train, y_train, X_test, y_test = coutils.data.cifar10()

       # Move all the data to the GPU
       X_train = X_train.cuda()
       y_train = y_train.cuda()
       X_test = X_test.cuda()
       y_test = y_test.cuda()

       # 0. Visualize some examples from the dataset.
       class_names = [
           'plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck'
       ]
       img = coutils.utils.visualize_dataset(X_train, y_train, 12, class_names)
       plt.imshow(img)
       plt.axis('off')
       plt.show()

       # 1. Normalize the data: subtract the mean RGB (zero mean)
       mean_image = X_train.mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).
      ↪mean(dim=3, keepdim=True)
       X_train -= mean_image
       X_test -= mean_image

       # 2. Reshape the image data into rows
       X_train = X_train.reshape(X_train.shape[0], -1)
       X_test = X_test.reshape(X_test.shape[0], -1)

       # 3. Add bias dimension and transform into columns
       ones_train = torch.ones(X_train.shape[0], 1, device=X_train.device)
       X_train = torch.cat([X_train, ones_train], dim=1)
       ones_test = torch.ones(X_test.shape[0], 1, device=X_test.device)
       X_test = torch.cat([X_test, ones_test], dim=1)

       # 4. Carve out part of the training set to use for validation.
       # For random permutation, you can use torch.randperm or torch.randint
```

```
    # But, for this homework, we use slicing instead.
    num_training = int( X_train.shape[0] * (1.0 - validation_ratio) )
    num_validation = X_train.shape[0] - num_training

    # Return the dataset as a dictionary
    data_dict = {}
    data_dict['X_val'] = X_train[num_training:num_training + num_validation]
    data_dict['y_val'] = y_train[num_training:num_training + num_validation]
    data_dict['X_train'] = X_train[0:num_training]
    data_dict['y_train'] = y_train[0:num_training]

    data_dict['X_test'] = X_test
    data_dict['y_test'] = y_test
    return data_dict

# Invoke the above function to get our data.
data_dict = get_CIFAR10_data()
print('Train data shape: ', data_dict['X_train'].shape)
print('Train labels shape: ', data_dict['y_train'].shape)
print('Validation data shape: ', data_dict['X_val'].shape)
print('Validation labels shape: ', data_dict['y_val'].shape)
print('Test data shape: ', data_dict['X_test'].shape)
print('Test labels shape: ', data_dict['y_test'].shape)
```



```
Train data shape:  torch.Size([49000, 3073])
Train labels shape:  torch.Size([49000])
Validation data shape:  torch.Size([1000, 3073])
Validation labels shape:  torch.Size([1000])
Test data shape:  torch.Size([10000, 3073])
Test labels shape:  torch.Size([10000])
```

For Softmax and SVM, we will analytically compute the gradient, as a sanity check.

```
[ ]: def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-7):
       """
       Utility function to perform numeric gradient checking. We use the centered
       difference formula to compute a numeric derivative:

       f'(x) =~ (f(x + h) - f(x - h)) / (2h)

       Rather than computing a full numeric gradient, we sparsely sample a few
       dimensions along which to compute numeric derivatives.

       Inputs:
```

```
  - f: A function that inputs a torch tensor and returns a torch scalar
  - x: A torch tensor giving the point at which to evaluate the numeric gradient
  - analytic_grad: A torch tensor giving the analytic gradient of f at x
  - num_checks: The number of dimensions along which to check
  - h: Step size for computing numeric derivatives
  """
  # fix random seed for
  coutils.utils.fix_random_seed()

  for i in range(num_checks):

    ix = tuple([random.randrange(m) for m in x.shape])

    oldval = x[ix].item()
    x[ix] = oldval + h # increment by h
    fxph = f(x).item() # evaluate f(x + h)
    x[ix] = oldval - h # increment by h
    fxmh = f(x).item() # evaluate f(x - h)
    x[ix] = oldval     # reset

    grad_numerical = (fxph - fxmh) / (2 * h)
    grad_analytic = analytic_grad[ix]
    rel_error_top = abs(grad_numerical - grad_analytic)
    rel_error_bot = (abs(grad_numerical) + abs(grad_analytic) + 1e-12)
    rel_error = rel_error_top / rel_error_bot
    msg = 'numerical: %f analytic: %f, relative error: %e'
    print(msg % (grad_numerical, grad_analytic, rel_error))
```

1.3 SVM Classifier

In this section, 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

[ ]:
```
def svm_loss_naive(W, X, y, reg):
  """
  Structured SVM loss function, naive implementation (with loops).

  Inputs have dimension D, there are C classes, and we operate on minibatches
  of N examples. When you implment the regularization over W, please DO NOT
  multiply the regularization term by 1/2 (no coefficient).

  Inputs:
```

```
- W: A PyTorch tensor of shape (D, C) containing weights.
- X: A PyTorch tensor of shape (N, D) containing a minibatch of data.
- y: A PyTorch tensor of shape (N,) containing training labels; y[i] = c means
  that X[i] has label c, where 0 <= c < C.
- reg: (float) regularization strength

Returns a tuple of:
- loss as torch scalar
- gradient of loss with respect to weights W; a tensor of same shape as W
"""
dW = torch.zeros_like(W) # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
  scores = W.t().mv(X[i])
  correct_class_score = scores[y[i]]
  for j in range(num_classes):
    if j == y[i]:
      continue
    margin = scores[j] - correct_class_score + 1 # note delta = 1
    if margin > 0:
      loss += margin
      #######################################################################
      # TODO:                                                               #
      # Compute the gradient of the loss function and store it dW. (part 1) #
      # Rather that first computing the loss and then computing the         #
      # derivative, it is simple to compute the derivative at the same time #
      # that the loss is being computed.                                    #
      #######################################################################
      # Replace "pass" statement with your code
      dW[:,y[i]]+=-X[i,:].t()
      dW[:,j]+=X[i,:].t()
      #######################################################################
      #                        END OF YOUR CODE                             #
      #######################################################################


# Right now the loss is a sum over all training examples, but we want it
# to be an average instead so we divide by num_train.
loss /= num_train

# Add regularization to the loss.
loss += reg * torch.sum(W * W)
```

```
  #########################################################################
  # TODO:                                                                 #
  # Compute the gradient of the loss function and store it in dW. (part 2)    #
  #########################################################################
  # Replace "pass" statement with your code
  dW=dW/num_train+reg*2*W
  #########################################################################
  #                             END OF YOUR CODE                          #
  #########################################################################

  return loss, dW
```

Evaluate the naive implementation of the loss we provided for you. You will get around 9.000175.

[ ]:
```
# generate a random SVM weight tensor of small numbers
coutils.utils.fix_random_seed()
W = torch.randn(3073, 10, device=data_dict['X_val'].device) * 0.0001

loss, grad = svm_loss_naive(W, data_dict['X_val'], data_dict['y_val'], 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.000433
```

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 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 (The relative errors should be less than `1e-6`).

[ ]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Use a random W and a minibatch of data from the val set for gradient checking
# For numeric gradient checking it is a good idea to use 64-bit floating point
# numbers for increased numeric precision; however when actually training models
# we usually use 32-bit floating point numbers for increased speed.
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()
batch_size = 64
X_batch = data_dict['X_val'][:64].double()
y_batch = data_dict['y_val'][:64]

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W.double(), X_batch, y_batch, reg=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.
f = lambda w: svm_loss_naive(w, X_batch, y_batch, reg=0.0)[0]
grad_numerical = grad_check_sparse(f, W.double(), grad)
```

```
numerical: -0.034577 analytic: -0.034577, relative error: 2.372452e-07
numerical: 0.126951 analytic: 0.126951, relative error: 5.721190e-08
numerical: -0.068597 analytic: -0.068597, relative error: 2.249695e-07
numerical: 0.025717 analytic: 0.025717, relative error: 4.774223e-07
numerical: 0.048266 analytic: 0.048266, relative error: 2.668362e-07
numerical: 0.052260 analytic: 0.052260, relative error: 2.475153e-07
numerical: 0.096133 analytic: 0.096133, relative error: 4.690208e-09
numerical: 0.032702 analytic: 0.032702, relative error: 3.644517e-07
numerical: -0.117158 analytic: -0.117158, relative error: 4.006759e-08
numerical: -0.154093 analytic: -0.154093, relative error: 7.809949e-08
```

Let's do the gradient check once again with regularization turned on. (You didn't forget the regularization gradient, did you?)

You should see relative errors less than `1e-5`.

[ ]:
```
# Use a minibatch of data from the val set for gradient checking
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()
batch_size = 64
X_batch = data_dict['X_val'][:64].double()
y_batch = data_dict['y_val'][:64]

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W.double(), X_batch, y_batch, reg=1e3)

# 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.
f = lambda w: svm_loss_naive(w, X_batch, y_batch, reg=1e3)[0]
grad_numerical = grad_check_sparse(f, W.double(), grad)
```

```
numerical: -0.121624 analytic: -0.121624, relative error: 7.160875e-08
numerical: 0.020923 analytic: 0.020923, relative error: 3.331613e-07
numerical: -0.076604 analytic: -0.076604, relative error: 1.800879e-07
numerical: 0.256069 analytic: 0.256069, relative error: 6.121908e-08
numerical: -0.330089 analytic: -0.330089, relative error: 4.165267e-08
numerical: 0.004713 analytic: 0.004713, relative error: 3.512375e-06
numerical: 0.101968 analytic: 0.101968, relative error: 1.802643e-08
numerical: 0.097235 analytic: 0.097235, relative error: 1.618033e-07
numerical: -0.117112 analytic: -0.117112, relative error: 2.948518e-08
numerical: -0.257260 analytic: -0.257260, relative error: 4.474401e-08
```

Now, let's implement vectorized version of SVM: `svm_loss_vectorized`. It should compute the same inputs and outputs as the naive version before, but it should involve **no explicit loops**.

```
[ ]: def svm_loss_vectorized(W, X, y, reg):
       """
       Structured SVM loss function, vectorized implementation. When you implment
       the regularization over W, please DO NOT multiply the regularization term by
       1/2 (no coefficient).

       Inputs and outputs are the same as svm_loss_naive.
       """
       loss = 0.0
       dW = torch.zeros_like(W) # initialize the gradient as zero

       #############################################################################
       # TODO:                                                                     #
       # Implement a vectorized version of the structured SVM loss, storing the    #
       # result in loss.                                                           #
       #############################################################################
       # Replace "pass" statement with your code
     #   print(X.shape)
     #   print(W.shape)
       scores = X.mm(W)
       num_train=X[0]

       correct_label_score_idxes = (range(scores.shape[0]), y)

       correct_label_scores = scores[correct_label_score_idxes]
       scores_diff = scores - torch.reshape(correct_label_scores, (-1, 1))

       scores_diff += 1

       scores_diff[correct_label_score_idxes] = 0
       indexes_of_neg_nums = torch.nonzero(scores_diff < 0)
       scores_diff[indexes_of_neg_nums] = 0

       loss = scores_diff.sum()
       num_train = X.shape[0]
       loss /= num_train
       # add in the regularization part.
       loss += reg * (W * W).sum()
       #############################################################################
       #                              END OF YOUR CODE                             #
       #############################################################################


       #############################################################################
       # TODO:                                                                     #
       # Implement a vectorized version of the gradient for the structured SVM     #
       # loss, storing the result in dW.                                           #
```

```
  #                                                                       #
  # Hint: Instead of computing the gradient from scratch, it may be easier #
  # to reuse some of the intermediate values that you used to compute the  #
  # loss.                                                                 #
  #########################################################################
  # Replace "pass" statement with your code
  scores_diff[scores_diff > 0] = 1
  correct_label_vals = torch.sum(scores_diff , 1) * -1
  scores_diff[correct_label_score_idxes] = correct_label_vals

  dW = X.t().mm(scores_diff)
  dW /= num_train
  # add the regularization contribution to the gradient
  dW += reg * 2* W
  #########################################################################
  #                             END OF YOUR CODE                          #
  #########################################################################

  return loss, dW
```

Let's first check the speed and performance bewteen the non-vectorized and the vectorized version. You should see a speedup of more than 100x.

(Note: It may have some difference, but should be less than 1e-6)

[ ]:
```
# Next implement the function svm_loss_vectorized; for now only compute the␣
 ↪loss;
# we will implement the gradient in a moment.

# Use random weights and a minibatch of val data for gradient checking
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()
X_batch = data_dict['X_val'][:128].double()
y_batch = data_dict['y_val'][:128]
reg = 0.000005

# Run and time the naive version
torch.cuda.synchronize()
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_naive = 1000.0 * (toc - tic)
print('Naive loss: %e computed in %.2fms' % (loss_naive, ms_naive))

# Run and time the vectorized version
torch.cuda.synchronize()
tic = time.time()
```

```
loss_vec, _ = svm_loss_vectorized(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_vec = 1000.0 * (toc - tic)
print('Vectorized loss: %e computed in %.2fms' % (loss_vec, ms_vec))

# The losses should match but your vectorized implementation should be much␣
 ↪faster.
print('Difference: %.2e' % (loss_naive - loss_vec))
print('Speedup: %.2fX' % (ms_naive / ms_vec))
```

```
Naive loss: 9.000144e+00 computed in 244.46ms
Vectorized loss: 9.000144e+00 computed in 6.26ms
Difference: -1.07e-14
Speedup: 39.05X
```

Then, let's compute the gradient of the loss function. We can check the difference of gradient as well. (The error should be less than 1e-6)

Now implement a vectorized version of the gradient computation in `svm_loss_vectorize` above. Run the cell below to compare the gradient of your naive and vectorized implementations. The difference between the gradients should be less than `1e-6`, and the vectorized version should run at least 100x faster.

[ ]:
```
# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.

# Use random weights and a minibatch of val data for gradient checking
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()
X_batch = data_dict['X_val'][:128].double()
y_batch = data_dict['y_val'][:128]
reg = 0.000005

# Run and time the naive version
torch.cuda.synchronize()
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_batch, y_batch, 0.000005)
torch.cuda.synchronize()
toc = time.time()
ms_naive = 1000.0 * (toc - tic)
print('Naive loss and gradient: computed in %.2fms' % ms_naive)

# Run and time the vectorized version
torch.cuda.synchronize()
tic = time.time()
_, grad_vec = svm_loss_vectorized(W, X_batch, y_batch, 0.000005)
torch.cuda.synchronize()
```

```
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 tensor, so
# we use the Frobenius norm to compare them.
grad_difference = torch.norm(grad_naive - grad_vec, p='fro')
print('Gradient difference: %.2e' % grad_difference)
print('Speedup: %.2fX' % (ms_naive / ms_vec))
```

```
Naive loss and gradient: computed in 225.95ms
Vectorized loss and gradient: computed in 0.002955s
Gradient difference: 0.00e+00
Speedup: 36.10X
```

Now that we have an efficient vectorized implementation of the SVM loss and its gradient, we can implement a training pipeline for linear classifiers.

Complete the implementation of the following function:

```
[ ]: def train_linear_classifier(loss_func, W, X, y, learning_rate=1e-3,
                                 reg=1e-5, num_iters=100, batch_size=200,␣
       ↪verbose=False):
       """
       Train this linear classifier using stochastic gradient descent.

       Inputs:
       - loss_func: loss function to use when training. It should take W, X, y
         and reg as input, and output a tuple of (loss, dW)
       - W: A PyTorch tensor of shape (D, C) giving the initial weights of the
         classifier. If W is None then it will be initialized here.
       - X: A PyTorch tensor of shape (N, D) containing training data; there are N
         training samples each of dimension D.
       - y: A PyTorch tensor of shape (N,) containing training labels; y[i] = c
         means that X[i] has label 0 <= c < C for C classes.
       - learning_rate: (float) learning rate for optimization.
       - reg: (float) regularization strength.
       - num_iters: (integer) number of steps to take when optimizing
       - batch_size: (integer) number of training examples to use at each step.
       - verbose: (boolean) If true, print progress during optimization.

       Returns: A tuple of:
       - W: The final value of the weight matrix and the end of optimization
       - loss_history: A list of Python scalars giving the values of the loss at each
         training iteration.
       """
       # assume y takes values 0...K-1 where K is number of classes
       num_classes = torch.max(y) + 1
```

```python
num_train, dim = X.shape
if W is None:
  # lazily initialize W
  W = 0.000001 * torch.randn(dim, num_classes, device=X.device, dtype=X.dtype)

# Run stochastic gradient descent to optimize W
loss_history = []
for it in range(num_iters):
  X_batch = None
  y_batch = None
  #########################################################################
  # TODO:                                                                 #
  # Sample batch_size elements from the training data and their           #
  # corresponding labels to use in this round of gradient descent.        #
  # Store the data in X_batch and their corresponding labels in           #
  # y_batch; after sampling, X_batch should have shape (batch_size, dim)  #
  # and y_batch should have shape (batch_size,)                           #
  #                                                                       #
  # Hint: Use torch.randint to generate indices.                          #
  #########################################################################
  # Replace "pass" statement with your code
  batch_idxes = torch.randint(num_train, (batch_size,))
  X_batch = X[batch_idxes, :]
  y_batch = y[batch_idxes]
  #########################################################################
  #                       END OF YOUR CODE                                #
  #########################################################################

  # evaluate loss and gradient
  loss, grad = loss_func(W, X_batch, y_batch, reg)
  loss_history.append(loss.item())

  # perform parameter update
  #########################################################################
  # TODO:                                                                 #
  # Update the weights using the gradient and the learning rate.          #
  #########################################################################
  # Replace "pass" statement with your code
  W -= learning_rate * grad
  #########################################################################
  #                       END OF YOUR CODE                                #
  #########################################################################

  if verbose and it % 100 == 0:
    print('iteration %d / %d: loss %f' % (it, num_iters, loss))

return W, loss_history
```

Once you have implemented the training function, run the following cell to train a linear classifier using some default hyperparameters:

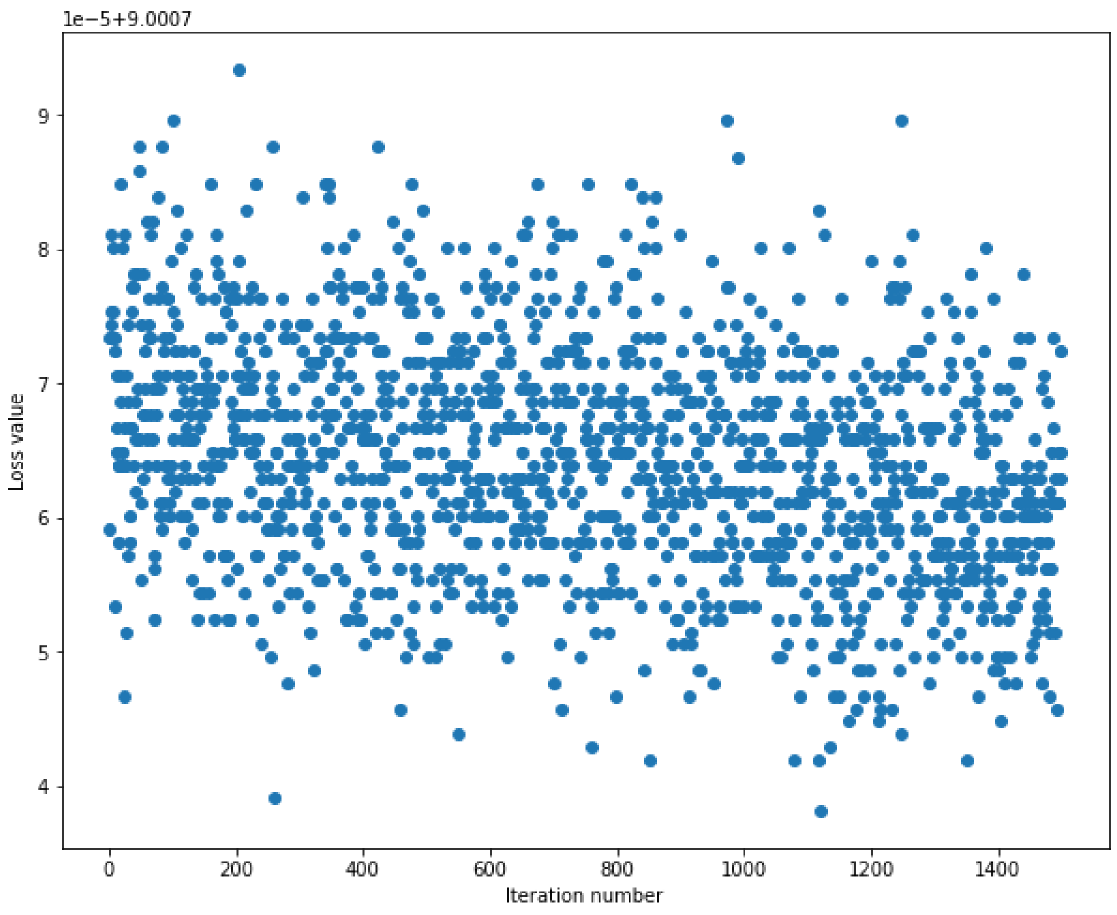
(You should see a final loss close to 9.0, and your training loop should run in about two seconds)

[ ]:
```
# fix random seed before we perform this operation
coutils.utils.fix_random_seed()

torch.cuda.synchronize()
tic = time.time()

W, loss_hist = train_linear_classifier(svm_loss_vectorized, None,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=3e-11, reg=2.5e4,
                                       num_iters=1500, verbose=True)

torch.cuda.synchronize()
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 9.000759
iteration 100 / 1500: loss 9.000790
iteration 200 / 1500: loss 9.000771
iteration 300 / 1500: loss 9.000764
iteration 400 / 1500: loss 9.000766
iteration 500 / 1500: loss 9.000767
iteration 600 / 1500: loss 9.000770
iteration 700 / 1500: loss 9.000748
iteration 800 / 1500: loss 9.000766
iteration 900 / 1500: loss 9.000761
iteration 1000 / 1500: loss 9.000764
iteration 1100 / 1500: loss 9.000766
iteration 1200 / 1500: loss 9.000779
iteration 1300 / 1500: loss 9.000757
iteration 1400 / 1500: loss 9.000749
That took 1.385345s
```

A useful debugging strategy is to plot the loss as a function of iteration number. In this case it seems our hyperparameters are not good, since the training loss is not decreasing very fast.

[ ]:
```
plt.plot(loss_hist, 'o')
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



Let's move on to the prediction stage:

```
[ ]: def predict_linear_classifier(W, X):
         """
         Use the trained weights of this linear classifier to predict labels for
         data points.

         Inputs:
         - W: A PyTorch tensor of shape (D, C), containing weights of a model
         - X: A PyTorch tensor of shape (N, D) containing training data; there are N
           training samples each of dimension D.

         Returns:
         - y_pred: PyTorch int64 tensor of shape (N,) giving predicted labels for each
           elemment of X. Each element of y_pred should be between 0 and C - 1.
         """
         y_pred = torch.zeros(X.shape[0])
         ###########################################################################
         # TODO:                                                                   #
```

```
  # Implement this method. Store the predicted labels in y_pred.          #
  #########################################################################
  # Replace "pass" statement with your code
  class_preds = X @ W
  y_pred = torch.argmax(class_preds, dim=1)
  #########################################################################
  #                          END OF YOUR CODE                             #
  #########################################################################
  return y_pred
```

Then, let's evaluate the performance our trained model on both the training and validation set. You should see validation accuracy less than 10%.

[ ]:
```
# evaluate the performance on both the training and validation set
y_train_pred = predict_linear_classifier(W, data_dict['X_train'])
train_acc = 100.0 * (data_dict['y_train'] == y_train_pred).float().mean().item()
print('Training accuracy: %.2f%%' % train_acc)
y_val_pred = predict_linear_classifier(W, data_dict['X_val'])
val_acc = 100.0 * (data_dict['y_val'] == y_val_pred).float().mean().item()
print('Validation accuracy: %.2f%%' % val_acc)
```

```
Training accuracy: 10.24%
Validation accuracy: 10.20%
```

Unfortunately, the performance of our initial model is quite bad. To find a better hyperparamters, let's first modulize the functions that we've implemented.

[ ]:
```
# Note: We will re-use `LinearClassifier' in Softmax section
class LinearClassifier(object):

  def __init__(self):
    self.W = None

  def train(self, X_train, y_train, learning_rate=1e-3, reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
    train_args = (self.loss, self.W, X_train, y_train, learning_rate, reg,
                  num_iters, batch_size, verbose)
    self.W, loss_history = train_linear_classifier(*train_args)
    return loss_history

  def predict(self, X):
    return predict_linear_classifier(self.W, X)

  def loss(self, W, X_batch, y_batch, reg):
    """
    Compute the loss function and its derivative.
    Subclasses will override this.

    Inputs:
```

```
    - W: A PyTorch tensor of shape (D, C) containing (trained) weight of a␣
↪model.
    - X_batch: A PyTorch tensor of shape (N, D) containing a minibatch of N
      data points; each point has dimension D.
    - y_batch: A PyTorch tensor of shape (N,) containing labels for the␣
↪minibatch.
    - reg: (float) regularization strength.

    Returns: A tuple containing:
    - loss as a single float
    - gradient with respect to self.W; an tensor of the same shape as W
    """
    pass
  def _loss(self, X_batch, y_batch, reg):
    self.loss(self.W, X_batch, y_batch, reg)


class LinearSVM(LinearClassifier):
  """ A subclass that uses the Multiclass SVM loss function """

  def loss(self, W, X_batch, y_batch, reg):
    return svm_loss_vectorized(W, X_batch, y_batch, reg)
```

1.4 Softmax Classifier

Similar to the SVM, you will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with 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

First, let's start from implementing the naive softmax loss function with nested loops.

[ ]:
```
def softmax_loss_naive(W, X, y, reg):
  """
  Softmax loss function, naive implementation (with loops).  When you implment
  the regularization over W, please DO NOT multiply the regularization term by
  1/2 (no coefficient).

  Inputs have dimension D, there are C classes, and we operate on minibatches
  of N examples.

  Inputs:
  - W: A PyTorch tensor of shape (D, C) containing weights.
  - X: A PyTorch tensor of shape (N, D) containing a minibatch of data.
  - y: A PyTorch tensor of shape (N,) containing training labels; y[i] = c means
    that X[i] has label c, where 0 <= c < C.
  - reg: (float) regularization strength

  Returns a tuple of:
  - loss as single float
  - gradient with respect to weights W; an tensor of same shape as W
  """
  # Initialize the loss and gradient to zero.
  loss = 0.0
  dW = torch.zeros_like(W)

  #############################################################################
  # TODO: Compute the softmax loss and its gradient using explicit loops.     #
  # Store the loss in loss and the gradient in dW. If you are not careful     #
  # here, it is easy to run into numeric instability (Check Numeric Stability #
  # in http://cs231n.github.io/linear-classify/). Plus, don't forget the      #
  # regularization!                                                           #
  #############################################################################

  N = len(X)
  D = len(W)
  C = len(W[0])
  for i in range(N):
    y_probs = W.T @ X[i]
    y_max = y_probs.max()
    y_probs -= y_max # numerical stability
    y_probs = torch.exp(y_probs)
    y_probs /= y_probs.sum()
    c = y[i]
    y_loss = -torch.log(y_probs[c])
    loss += y_loss

    dW += X[i, :, None] * y_probs[None, :]
    dW[:, c] -= X[i, :]

  loss /= N
  dW /= N

  loss += reg * W.square().sum()
  dW += 2 * reg * W

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

  return loss, dW
```

As a sanity check to see whether we have implemented the loss correctly, run the softmax classifier with a small random weight matrix and no regularization. You should see loss near $\log(10) = 2.3$

[ ]:
```
# Generate a random softmax weight tensor and use it to compute the loss.
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()

X_batch = data_dict['X_val'][:128].double()
y_batch = data_dict['y_val'][:128]

# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_batch, y_batch, reg=0.0)

# As a rough sanity check, our loss should be something close to log(10.0).
print('loss: %f' % loss)
print('sanity check: %f' % (math.log(10.0)))
```

```
loss: 2.302600
sanity check: 2.302585
```

Next, we use gradient checking to debug the analytic gradient of our naive softmax loss function. If you've implemented the gradient correctly, you should see relative errors less than `1e-6`.

[ ]:
```
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()
X_batch = data_dict['X_val'][:128].double()
y_batch = data_dict['y_val'][:128]

loss, grad = softmax_loss_naive(W, X_batch, y_batch, reg=0.0)

f = lambda w: softmax_loss_naive(w, X_batch, y_batch, reg=0.0)[0]
grad_check_sparse(f, W, grad, 10)
```

```
numerical: 0.008387 analytic: 0.008387, relative error: 1.366549e-07
numerical: 0.009227 analytic: 0.009227, relative error: 1.339656e-07
numerical: -0.002471 analytic: -0.002471, relative error: 1.718331e-07
numerical: -0.003144 analytic: -0.003144, relative error: 2.403080e-06
numerical: 0.006011 analytic: 0.006011, relative error: 6.813253e-08
numerical: 0.005936 analytic: 0.005936, relative error: 2.473992e-07
numerical: 0.015703 analytic: 0.015703, relative error: 2.149831e-08
numerical: 0.006452 analytic: 0.006452, relative error: 2.068055e-09
numerical: -0.015533 analytic: -0.015533, relative error: 2.569959e-07
numerical: -0.010170 analytic: -0.010170, relative error: 4.657956e-07
```

Let's perform another gradient check with regularization enabled. Again you should see relative errors less than `1e-6`.

```
[ ]: coutils.utils.fix_random_seed()
     W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device).double()
     reg = 10.0

     X_batch = data_dict['X_val'][:128].double()
     y_batch = data_dict['y_val'][:128]

     loss, grad = softmax_loss_naive(W, X_batch, y_batch, reg)

     f = lambda w: softmax_loss_naive(w, X_batch, y_batch, reg)[0]
     grad_check_sparse(f, W, grad, 10)
```

```
numerical: 0.007517 analytic: 0.007517, relative error: 2.286123e-07
numerical: 0.008167 analytic: 0.008167, relative error: 1.238116e-07
numerical: -0.002551 analytic: -0.002551, relative error: 3.147037e-08
numerical: -0.000841 analytic: -0.000841, relative error: 8.976250e-06
numerical: 0.002228 analytic: 0.002228, relative error: 4.070641e-07
numerical: 0.005460 analytic: 0.005460, relative error: 1.891176e-07
numerical: 0.015762 analytic: 0.015762, relative error: 7.076879e-08
numerical: 0.007097 analytic: 0.007097, relative error: 1.474077e-07
numerical: -0.015532 analytic: -0.015532, relative error: 2.149161e-07
numerical: -0.011201 analytic: -0.011201, relative error: 3.391712e-07
```

Then, let's move on to the vectorized form

```
[ ]: def softmax_loss_vectorized(W, X, y, reg):
       """
       Softmax loss function, vectorized version.  When you implment the
       regularization over W, please DO NOT multiply the regularization term by 1/2
       (no coefficient).

       Inputs and outputs are the same as softmax_loss_naive.
       """
       # Initialize the loss and gradient to zero.
       loss = 0.0
       dW = torch.zeros_like(W)

       ##############################################################################
       # TODO: Compute the softmax loss and its gradient using no explicit loops.  #
       # Store the loss in loss and the gradient in dW. If you are not careful     #
       # here, it is easy to run into numeric instability (Check Numeric Stability #
       # in http://cs231n.github.io/linear-classify/). Don't forget the            #
       # regularization!                                                           #
       ##############################################################################
       # Replace "pass" statement with your code
       N = len(X)
       D = len(W)
       C = len(W[0])
```

```
  y_probs = X @ W
  y_max = y_probs.max(-1, keepdim=True)
  y_probs -= y_max.values # numerical stability
  y_probs = torch.exp(y_probs)
  y_probs /= y_probs.sum(-1, keepdim=True)
  loss -= torch.log(y_probs[range(N), y]).sum()

  dW += X.T @ y_probs
  other_grad = torch.zeros_like(dW.T)
  other_grad.index_add_(0, y, X)
  dW -= other_grad.T

  loss /= N
  dW /= N

  loss += reg * W.square().sum()
  dW += 2 * reg * W
  #############################################################################
  #                              END OF YOUR CODE                             #
  #############################################################################

  return loss, dW
```

Now that we have a naive implementation of the softmax loss function and its gradient, implement a vectorized version in softmax_loss_vectorized. The two versions should compute the same results, but the vectorized version should be much faster.

The differences between the naive and vectorized losses and gradients should both be less than `1e-6`, and your vectorized implementation should be at least 100x faster than the naive implementation.

[ ]:
```
coutils.utils.fix_random_seed()
W = 0.0001 * torch.randn(3073, 10, device=data_dict['X_val'].device)
reg = 0.05

X_batch = data_dict['X_val'][:128]
y_batch = data_dict['y_val'][:128]

# Run and time the naive version
torch.cuda.synchronize()
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_naive = 1000.0 * (toc - tic)
print('naive loss: %e computed in %fs' % (loss_naive, ms_naive))

# Run and time the vectorized version
```

```
torch.cuda.synchronize()
tic = time.time()
loss_vec, grad_vec = softmax_loss_vectorized(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_vec = 1000.0 * (toc - tic)
print('vectorized loss: %e computed in %fs' % (loss_vec, ms_vec))

# we use the Frobenius norm to compare the two versions of the gradient.
loss_diff = (loss_naive - loss_vec).abs().item()
grad_diff = torch.norm(grad_naive - grad_vec, p='fro')
print('Loss difference: %.2e' % loss_diff)
print('Gradient difference: %.2e' % grad_diff)
print('Speedup: %.2fX' % (ms_naive / ms_vec))
```

```
naive loss: 2.302615e+00 computed in 30.057907s
vectorized loss: 2.302616e+00 computed in 1.835823s
Loss difference: 4.77e-07
Gradient difference: 3.51e-07
Speedup: 16.37X
```

Let's check that your implementation of the softmax loss is numerically stable.

If either of the following print `nan` then you should double-check the numeric stability of your implementations.

```
[ ]: device = data_dict['X_train'].device
dtype = torch.float32
D = data_dict['X_train'].shape[1]
C = 10

W_ones = torch.ones(D, C, device=device, dtype=dtype)
W, loss_hist = train_linear_classifier(softmax_loss_naive, W_ones,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-8, reg=2.5e4,
                                       num_iters=1, verbose=True)


W_ones = torch.ones(D, C, device=device, dtype=dtype)
W, loss_hist = train_linear_classifier(softmax_loss_vectorized, W_ones,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-8, reg=2.5e4,
                                       num_iters=1, verbose=True)
```

```
iteration 0 / 1: loss 768249984.000000
iteration 0 / 1: loss 768249984.000000
```

Now lets train a softmax classifier with some default hyperparameters:

[ ]:

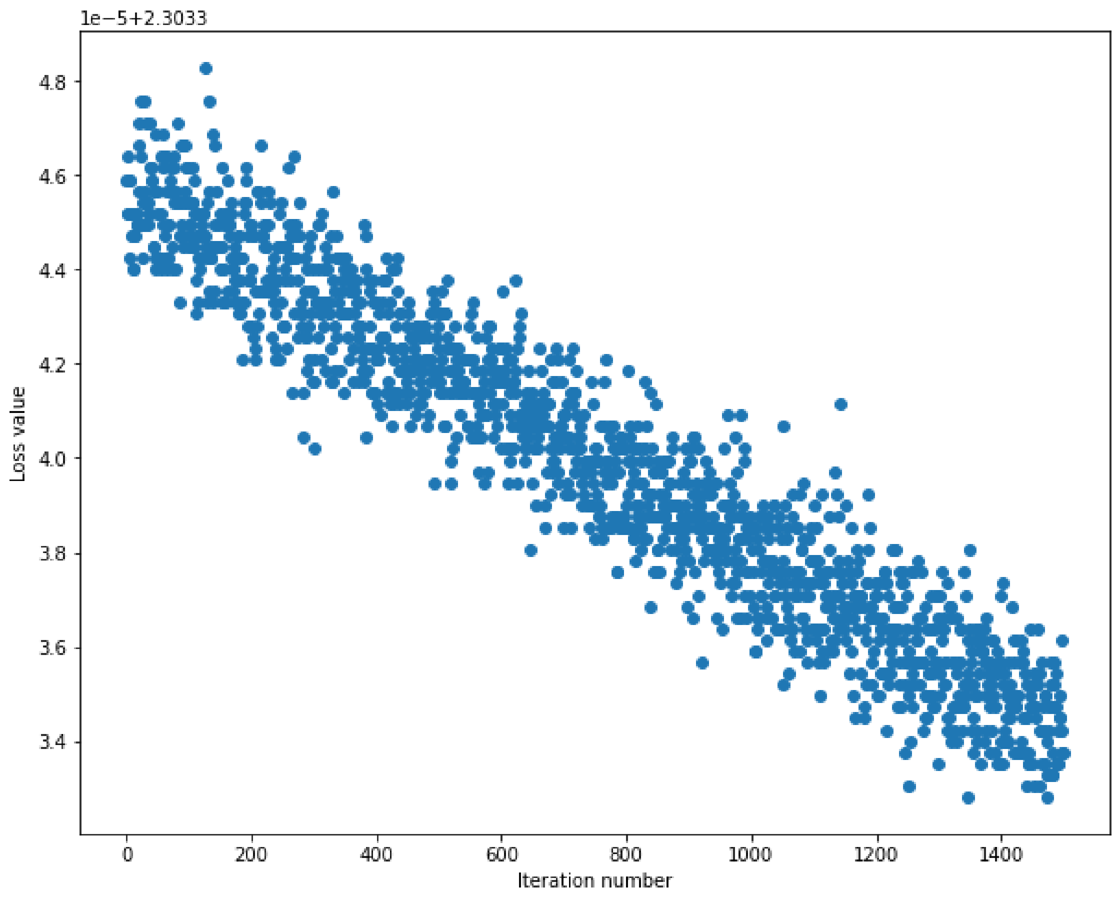
```
# fix random seed before we perform this operation
coutils.utils.fix_random_seed(10)

torch.cuda.synchronize()
tic = time.time()

W, loss_hist = train_linear_classifier(softmax_loss_vectorized, None,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-10, reg=2.5e4,
                                       num_iters=1500, verbose=True)

torch.cuda.synchronize()
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 2.303346
iteration 100 / 1500: loss 2.303344
iteration 200 / 1500: loss 2.303344
iteration 300 / 1500: loss 2.303342
iteration 400 / 1500: loss 2.303342
iteration 500 / 1500: loss 2.303342
iteration 600 / 1500: loss 2.303342
iteration 700 / 1500: loss 2.303341
iteration 800 / 1500: loss 2.303339
iteration 900 / 1500: loss 2.303339
iteration 1000 / 1500: loss 2.303339
iteration 1100 / 1500: loss 2.303336
iteration 1200 / 1500: loss 2.303335
iteration 1300 / 1500: loss 2.303337
iteration 1400 / 1500: loss 2.303334
That took 1.109919s
```

Plot the loss curve:

[ ]:

```
plt.plot(loss_hist, 'o')
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



Let's compute the accuracy of current model. It should be less than 10%.

```
[ ]: # evaluate the performance on both the training and validation set
y_train_pred = predict_linear_classifier(W, data_dict['X_train'])
train_acc = 100.0 * (data_dict['y_train'] == y_train_pred).float().mean().item()
print('training accuracy: %.2f%%' % train_acc)
y_val_pred = predict_linear_classifier(W, data_dict['X_val'])
val_acc = 100.0 * (data_dict['y_val'] == y_val_pred).float().mean().item()
print('validation accuracy: %.2f%%' % val_acc)
```

```
training accuracy: 8.43%
validation accuracy: 8.00%
```

Now 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.

To get full credit for the assignment, your best model found through cross-validation should achieve an accuracy above 0.37 on the validation set.

(Our best model was above 0.40 – did you beat us?)

```
[ ]: class Softmax(LinearClassifier):
       """ A subclass that uses the Softmax + Cross-entropy loss function """
       def loss(self, W, X_batch, y_batch, reg):
         return softmax_loss_vectorized(W, X_batch, y_batch, reg)
```

```
[ ]: results = {}
     best_val = -1
     best_softmax = None

     learning_rates = [1e-1, 1e-2, 1e-3, 1e-4] # learning rate candidates
     regularization_strengths = [1e-1, 1e-2, 1e-3] # regularization strengths␣
      ↪candidates

     # As before, store your cross-validation results in this dictionary.
     # The keys should be tuples of (learning_rate, regularization_strength) and
     # the values should be tuples (train_accuracy, val_accuracy)
     results = {}

     ################################################################################
     # TODO:                                                                        #
     # Use the validation set to set the learning rate and regularization strength. #
     # This should be similar to the cross-validation that you used for the SVM,    #
     # but you may need to select different hyperparameters to achieve good         #
     # performance with the softmax classifier. Save your best trained softmax      #
     # classifer in best_softmax.                                                   ␣
      ↪#
     ################################################################################
     # Replace "pass" statement with your code
     grid_search = [ (lr,rg) for lr in learning_rates for rg in␣
      ↪regularization_strengths ]
     X_train=data_dict['X_train'].double()
     y_train=data_dict['y_train']
     X_val=data_dict['X_val'].double()
     y_val=data_dict['y_val']
     for lr, rg in grid_search:
         # Create a new SVM instance
         svm = Softmax()
         # Train the model with current parameters
         train_loss = svm.train(X_train, y_train, learning_rate=lr, reg=rg,
                          num_iters=2000,batch_size=200, verbose=False)
         # Predict values for training set
         y_train_pred = svm.predict(X_train)
         # Calculate accuracy
         train_accuracy = torch.mean((y_train_pred == y_train).float())
         # Predict values for validation set
         y_val_pred = svm.predict(X_val)
         # Calculate accuracy
```

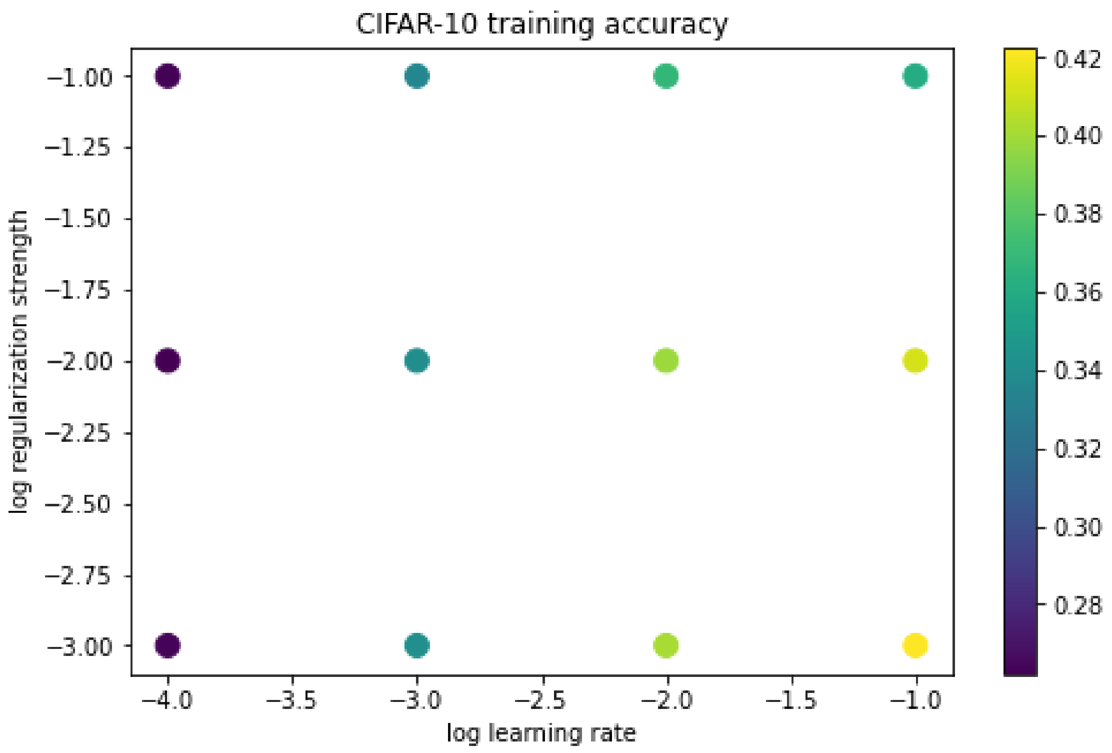
```
    val_accuracy = torch.mean((y_val_pred == y_val).float())
    # Save results
    results[(lr,rg)] = (train_accuracy, val_accuracy)
    if best_val < val_accuracy:
        best_val = val_accuracy
        best_softmax = svm
################################################################################
#                              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)
```

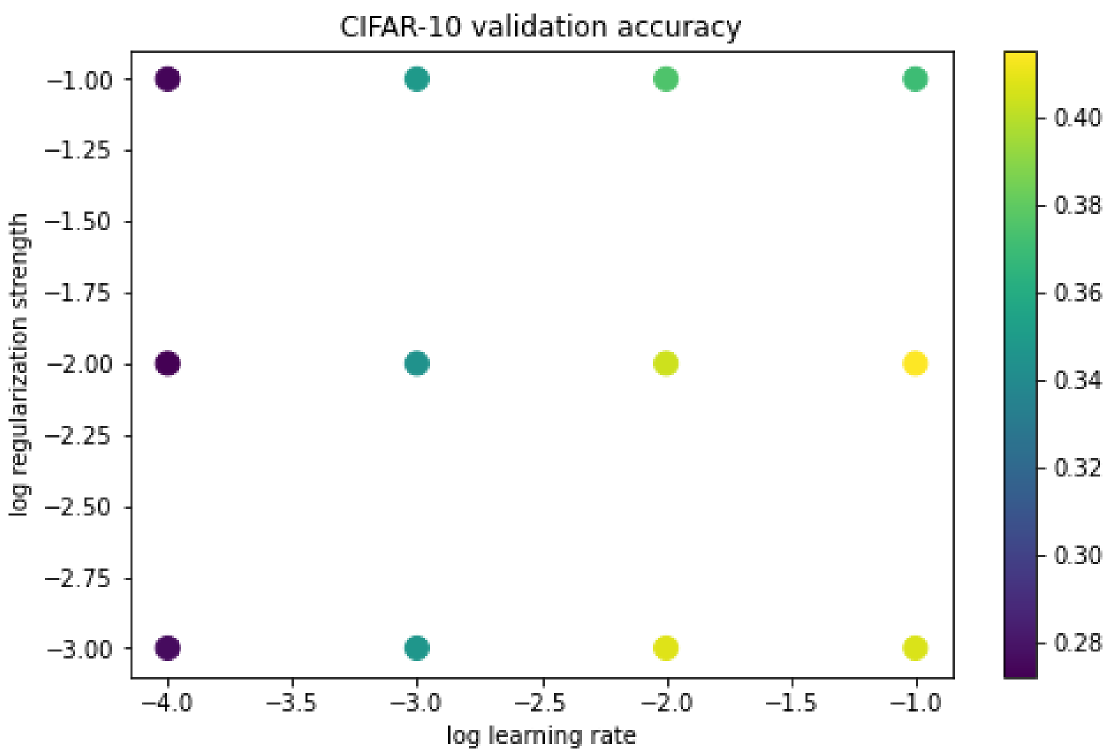
```
lr 1.000000e-04 reg 1.000000e-03 train accuracy: 0.263694 val accuracy: 0.276000
lr 1.000000e-04 reg 1.000000e-02 train accuracy: 0.262082 val accuracy: 0.272000
lr 1.000000e-04 reg 1.000000e-01 train accuracy: 0.262776 val accuracy: 0.274000
lr 1.000000e-03 reg 1.000000e-03 train accuracy: 0.341714 val accuracy: 0.347000
lr 1.000000e-03 reg 1.000000e-02 train accuracy: 0.340551 val accuracy: 0.345000
lr 1.000000e-03 reg 1.000000e-01 train accuracy: 0.335939 val accuracy: 0.349000
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.401714 val accuracy: 0.408000
lr 1.000000e-02 reg 1.000000e-02 train accuracy: 0.398143 val accuracy: 0.404000
lr 1.000000e-02 reg 1.000000e-01 train accuracy: 0.368102 val accuracy: 0.375000
lr 1.000000e-01 reg 1.000000e-03 train accuracy: 0.422571 val accuracy: 0.407000
lr 1.000000e-01 reg 1.000000e-02 train accuracy: 0.412082 val accuracy: 0.415000
lr 1.000000e-01 reg 1.000000e-01 train accuracy: 0.361082 val accuracy: 0.370000
best validation accuracy achieved during cross-validation: 0.415000
```

Run the following to visualize your cross-validation results:

```
[ ]: 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].cpu() for x in results]
     plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap='viridis')
     plt.colorbar()
     plt.xlabel('log learning rate')
     plt.ylabel('log regularization strength')
     plt.title('CIFAR-10 training accuracy')
     plt.gcf().set_size_inches(8, 5)
     plt.show()
```

```
# plot validation accuracy
colors = [results[x][1].cpu() for x in results] # default size of markers is 20
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap='viridis')
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.gcf().set_size_inches(8, 5)
plt.show()
```





Them, evaluate the performance of your best model on test set. To get full credit for this assignment you should achieve a test-set accuracy above 0.36.

(Our best was just over 0.40 – did you beat us?)

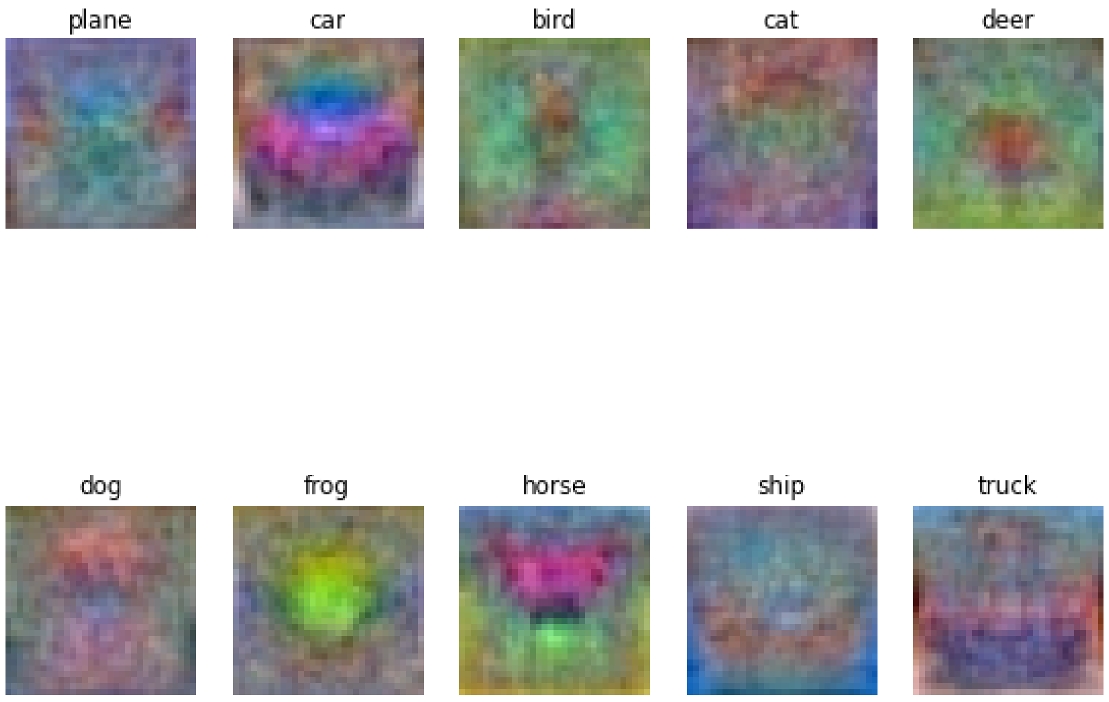
[ ]:
```
y_test_pred = best_softmax.predict(data_dict['X_test'].double())
test_accuracy = torch.mean((data_dict['y_test'] == y_test_pred).float())
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.403900
```

Finally, let's visualize the learned weights for each class

[ ]:
```
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(3, 32, 32, 10)
w = w.transpose(0, 2).transpose(1, 0)

w_min, w_max = torch.min(w), torch.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.type(torch.uint8).cpu())
plt.axis('off')
plt.title(classes[i])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck