import numpy as np

from random import shuffle

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.

Inputs:

- W: A numpy array of shape (D, C) containing weights.

- X: A numpy array of shape (N, D) containing a minibatch of data.

- y: A numpy array 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 array of same shape as W

"""

dW = np.zeros(W.shape) # 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 = X[i].dot(W)

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

dW[:, j] += X[i]

dW[:, y[i]] += -X[i]

# 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

dW /= num\_train

# Add regularization to the loss.

loss += 0.5 \* reg \* np.sum(W \* W)

dW += reg \* W

#############################################################################

# TODO: #

# Compute the gradient of the loss function and store it dW. #

# Rather that first computing the loss and then computing the derivative, #

# it may be simpler to compute the derivative at the same time that the #

# loss is being computed. As a result you may need to modify some of the #

# code above to compute the gradient. #

#############################################################################

return loss, dW

def svm\_loss\_vectorized(W, X, y, reg):

"""

Structured SVM loss function, vectorized implementation.

Inputs and outputs are the same as svm\_loss\_naive.

"""

loss = 0.0

dW = np.zeros(W.shape) # initialize the gradient as zero

num\_train = X.shape[0]

#############################################################################

# TODO: #

# Implement a vectorized version of the structured SVM loss, storing the #

# result in loss. #

#############################################################################

scores = X.dot(W)

margin = scores - scores[np.arange(num\_train), y].reshape(num\_train, 1) + 1

margin[np.arange(num\_train), y] = 0.0 #这一列不该计算，归零

margin = (margin > 0) \* margin

loss += margin.sum() / num\_train

loss += 0.5 \* reg \* np.sum(W \* W)

#############################################################################

# 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. #

#############################################################################

margin = (margin > 0) \* 1

row\_sum = np.sum(margin, axis=1) #

margin[np.arange(num\_train), y] = -row\_sum

dW = X.T.dot(margin)/num\_train + reg \* W

#############################################################################

# END OF YOUR CODE #

#############################################################################

return loss, dW