```
import numpy as np
from sklearn.model_selection import train_test_split
class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
        Initializes the weight matrix of the Softmax classifier.
        Note that it has shape (C, D) where C is the number of
        classes and D is the feature size.
        """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

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

    Inputs:
    - 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.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0

    # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the normalized softmax loss.  Store it as the variable loss.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
        #       training examples.)
    # ================================================================ #
    num_train = X.shape[0]
    for i in range(num_train):
      score = np.dot(X[i],self.W.T)
      score = score - np.max(score)
      loss = loss - score[y[i]]

      sm = 0
      for s in score:
        sm = sm + np.exp(s)
      loss = loss + np.log(sm)
    loss = loss / num_train

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss

  def loss_and_grad(self, X, y):
    """
        Same as self.loss(X, y), except that it also returns the gradient.

        Output: grad -- a matrix of the same dimensions as W containing
                the gradient of the loss with respect to W.
        """
```

```
    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)

    num_train = X.shape[0]
    num_class = self.W.shape[0]
    for i in range(num_train):
      score = np.dot(X[i],self.W.T)
      score = score - np.max(score)
      loss = loss - score[y[i]]

      sm = 0
      for s in score:
        sm = sm + np.exp(s)
      loss = loss + np.log(sm)
      for j in range(num_class):
        sftmx = np.exp(score[j]) / np.sum(np.exp(score))
        if j == y[i]:
          grad[j,:] = grad[j,:] + (sftmx - 1) * X[i]
        else:
          grad[j,:] = grad[j,:] + sftmx * X[i]
    loss = loss / num_train
    grad = grad / num_train
    # ================================================================ #
    # YOUR CODE HERE:
      #   Calculate the softmax loss and the gradient. Store the gradient
      #   as the variable grad.
    # ================================================================ #


    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
    """
    sample a few random elements and only return numerical
    in these dimensions.
    """

    for i in np.arange(num_checks):
      ix = tuple([np.random.randint(m) for m in self.W.shape])

      oldval = self.W[ix]
      self.W[ix] = oldval + h # increment by h
      fxph = self.loss(X, y)
      self.W[ix] = oldval - h # decrement by h
      fxmh = self.loss(X,y) # evaluate f(x - h)
      self.W[ix] = oldval # reset

      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs
(grad_analytic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,
grad_analytic, rel_error))

  def fast_loss_and_grad(self, X, y):
    """
    A vectorized implementation of loss_and_grad. It shares the same
      inputs and ouptuts as loss_and_grad.
    """
    loss = 0.0
```

```
    grad = np.zeros(self.W.shape) # initialize the gradient as zero
    num_train = X.shape[0]
    num_class = self.W.shape[0]
    # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the softmax loss and gradient WITHOUT any for loops.
    # ================================================================ #

    loss = -np.sum(np.log(np.true_divide(np.exp(np.sum(X * self.W[y],axis=1) - 
np.max(np.dot(X,self.W.T),axis=1)),np.sum(np.exp(np.dot(X,self.W.T) - np.max(np.dot
(X,self.W.T),axis=1).reshape(num_train,1)),axis=1))))
    #loss = np.sum([np.log(np.sum([np.exp(np.dot(X[i],self.W.T) - np.max(np.dot(X
[i],self.W.T)))])) - np.dot(X[i],self.W[y[i]].T) + np.max(np.dot(X[i],self.W.T)) 
for i in range(num_train)])
    loss = loss / num_train
    y_belong = np.argwhere(y == 0).T.tolist() + np.argwhere(y == 1).T.tolist() + 
np.argwhere(y == 2).T.tolist() + np.argwhere(y == 3).T.tolist() + np.argwhere(y == 
4).T.tolist() + np.argwhere(y == 5).T.tolist() + np.argwhere(y == 6).T.tolist() + 
np.argwhere(y == 7).T.tolist() + np.argwhere(y == 8).T.tolist() + np.argwhere(y == 
9).T.tolist()
    grad = np.dot(np.true_divide(np.exp(np.dot(X,self.W.T) - np.max(np.dot
(X,self.W.T),axis=1).reshape(num_train,1)),np.sum(np.exp(np.dot(X,self.W.T) - 
np.max(np.dot(X,self.W.T),axis=1).reshape(num_train,1)),axis=1).reshape
(num_train,1)).T,X) - [np.sum(X[y_belong[i]],axis=0) for i in range(num_class)]
    #grad = [sum([(np.exp(np.dot(X[i],self.W.T)[j] - np.max(np.dot(X
[i],self.W.T))) / np.sum(np.exp(np.dot(X[i],self.W.T) - np.max(np.dot(X
[i],self.W.T)))) - (0 if j != y[i] else 1)) * X[i] for i in range(num_train)]) for 
j in range(num_class)]
    grad = np.asarray(grad) / num_train

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.
    - y: A numpy array 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.
    - 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.

    Outputs:
    A list containing the value of the loss function at each training iteration.
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number 
of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights 
of self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []

    for it in np.arange(num_iters):
      X_batch = None
```

```
      y_batch = None

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #   gradient descent.  After sampling,
      #     - X_batch should have shape: (dim, batch_size)
          #     - y_batch should have shape: (batch_size,)
          #   The indices should be randomly generated to reduce correlations
          #   in the dataset.  Use np.random.choice.  It's okay to sample with
          #   replacement.
      # ================================================================ #
      _, X_batch,_, y_batch = train_test_split(X,y,test_size=batch_size/num_train,
random_state=int(np.random.randint(0,2**32-1,size=1)))
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

      # evaluate loss and gradient
      loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
      loss_history.append(loss)

      # ================================================================ #
      # YOUR CODE HERE:
      #   Update the parameters, self.W, with a gradient step
      # ================================================================ #
      self.W = self.W - learning_rate * grad

          # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

      if verbose and it % 100 == 0:
        print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

    return loss_history

  def predict(self, X):
    """
    Inputs:
    - X: N x D array of training data. Each row is a D-dimensional point.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
      class.
    """
    y_pred = np.zeros(X.shape[1])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ================================================================ #
    y_pred = np.argmax(np.dot(X,self.W.T),axis=1)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```