```python
import numpy as np


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.)
    # ================================================================ #
    weights = self.W
    losses = []

    for index in range(0,X.shape[0]):
        data = X[index]
        multiplication = weights.dot(data.T).T
        exponentials = np.exp(multiplication)
        exponentialSum = np.sum(exponentials)
        exponentials=exponentials/exponentialSum
        label = y[index]
        crossEntropy = -np.log(exponentials[label])
        losses.append(crossEntropy)
    loss = sum(losses)/len(losses)
    # ================================================================ #
    # END YOUR CODE HERE
```

```python
        # ================================================================ #

        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)

        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the softmax loss and the gradient. Store the gradient
        #   as the variable grad.
        # ================================================================ #
        weights = self.W
        losses=[]
        for index in range(0,X.shape[0]):
            data = X[index]
            multiplication = weights.dot(data.T).T
            exponentials = np.exp(multiplication)
            exponentialSum = np.sum(exponentials)
            exponentials=exponentials/exponentialSum
            label = y[index]
            crossEntropy = -np.log(exponentials[label])
            losses.append(crossEntropy)
            yHot = np.zeros([1,np.max(y)+1])
            yHot[0,label]=1
            data = np.reshape(data,[1,data.shape[0]])
            exponentials = np.reshape(exponentials,[1,exponentials.shape[0]])
            derivative = np.dot(data.T, (exponentials - yHot))
            grad=grad+derivative.T
        loss = sum(losses)/len(losses)
        grad = grad/[X.shape[0]]

        # ================================================================ #
        # 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_ana
        print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analyt

    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

      # ================================================================ #
      # YOUR CODE HERE:
      #   Calculate the softmax loss and gradient WITHOUT any for loops.
      # ================================================================ #
      weights = self.W
      multiplication =weights.dot(X.T).T
      multiplication = (multiplication.T - np.amax(multiplication,axis = 1)).T
      soft = np.exp(multiplication)
      sums = np.sum(soft,axis=1)
      probs = (soft.T / sums).T
      predsForClass = probs[np.arange(y.size),y]
      loss = -np.log(predsForClass+1e-10) #To aviod log(0)
      loss = np.mean(loss)
      yHot = np.zeros([y.shape[0],9+1])
      yHot[np.arange(y.size),y] = 1
      grad = (1/X.shape[0])*np.dot(X.T, (probs - yHot)).T

      # ================================================================ #
      # 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.
```

```python
    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 classe

    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.
      # ================================================================ #
      indices = np.random.choice(np.arange(num_train), batch_size)
      XBatch = X[indices]
      yBatch = y[indices]
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

      # evaluate loss and gradient
      loss, grad = self.fast_loss_and_grad(XBatch, yBatch)
      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.
# ================================================================ #
weights = self.W
multiplication =weights.dot(X.T).T
soft = np.exp(multiplication)
y_pred=np.argmax(soft,axis=1)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return y_pred
```