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

    num_samples = X.shape[0]
    num_classes = self.W.shape[0]

    for i in np.arange(num_samples):

      temp_log = 0
      for j in np.arange(num_classes):
        temp_log += np.exp(self.W[j].dot(X[i]))

      loss += np.log(temp_log) - self.W[y[i]].dot(X[i])

    loss = loss/num_samples

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

```python
  """

    # 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.
    # ================================================================ #

    num_samples = X.shape[0]
    num_classes = self.W.shape[0]

    for i in np.arange(num_samples):

      temp_log = 0
      temp_grad = np.zeros_like(self.W[:,0])
      for j in np.arange(num_classes):
        score = np.exp(self.W[j].dot(X[i]))
        temp_log += score
        temp_grad[j] = score

      temp_grad /= temp_log
      temp_grad[y[i]] -= 1
      grad += temp_grad[:, np.newaxis]*X[i]


      loss += np.log(temp_log) - self.W[y[i]].dot(X[i])

    loss = loss/num_samples
    grad = grad/num_samples


    # ================================================================ #
    # 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 ouputs 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.
    # ================================================================ #

    num_samples = X.shape[0]
    num_classes = self.W.shape[0]

    # Loss
    scores = self.W.dot(X.T)
    e_scores = np.exp(self.W.dot(X.T))
    sums = np.sum(e_scores, axis=0)
    log_sums = np.log(sums)
    y_terms = scores[y, np.arange(num_samples)]
    loss = np.sum(log_sums - y_terms)/num_samples


    # Grad
    e_scores = e_scores/sums
    e_scores[y,np.arange(num_samples)] -= 1
    grad = e_scores.dot(X)/num_samples

    # ================================================================ #
    # 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.
      # ================================================================ #
      mask = np.random.choice(np.arange(X.shape[0]), batch_size)
      X_batch = X[mask]
```

```
      y_batch = y[mask]
      # ================================================================ #
      # 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 -= 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.
    # ================================================================ #

    scores = self.W.dot(X.T)
    y_pred = np.argmax(scores,axis=0)



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

    return y_pred
```