```
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
    num_classes = self.W.shape[0]     # C = num_classes
    num_train = X.shape[0]

    exp_a = np.zeros((num_classes,num_train))
    # ================================================================ #
    # 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.)
    # ================================================================ #


    for i in np.arange(num_train):

        Loss = 0.0

        class_scores = np.dot(self.W,X[i,:].T)        # calculating class scores (C x 1 vector
        class_scores -= np.max(class_scores)          # considering the possible issue for nur

        exp_a[:,i] = np.exp(class_scores)                  # turning class scores to probabilit

        Loss -= np.log(exp_a[y[i],i]/np.sum(exp_a[:,i]))


        #p[:,i] = exp_a[:,i]/np.sum(exp_a[:,i])                  # p now is a valid probability
        #print(p[:,i])

        loss += Loss
        #print(Loss,i)

    pass
    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)
    grad_tmp = np.zeros_like(self.W)
    num_classes = self.W.shape[0]     # C = num_classes
    num_train = X.shape[0]

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ================================================================ #

    exp_a = np.zeros((num_classes,num_train))
    for i in np.arange(num_train):

        Loss = 0.0

        class_scores = np.dot(self.W,X[i,:].T)        # calculating class scores (C x 1 vector
        class_scores -= np.max(class_scores)          # considering the possible issue for nur

        exp_a[:,i] = np.exp(class_scores)                 # turning class scores to probabilit

        Loss -= np.log(exp_a[y[i],i]/np.sum(exp_a[:,i]))


        #if i==0:
        grada = np.zeros(X.shape[1])

        for j in range(num_classes):
            if j != y[i]:
                grad_tmp[j,:] = X[i,:].T * (exp_a[j,i] / np.sum(exp_a[:,i]))
            else:
                grad_tmp[j,:] = X[i,:].T * (exp_a[j,i] / np.sum(exp_a[:,i])) - X[i,:].T

        grad += grad_tmp
        loss += Loss

    pass


    loss /= num_train
    grad /= num_train
    # ================================================================ #
    # 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_analyt
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic,

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

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

#     # vectorized loss calculation #
    class_scores_matrix = np.dot(self.W,X.T)       # calculating class scores matrix (C x m):
    class_scores_matrix -= np.max(class_scores_matrix)      # considering the possible issue
    exp_a = np.exp(class_scores_matrix)          # calculating the exponents

#     y_exp = np.array(exp_a[y, np.arange(0, class_scores_matrix.shape[1])])
#     #print(exp_a[:,:3])
#     #print(y[:3])
#     #print(y_exp[:3])

#     tt = np.sum(exp_a,axis=0)
#     tt2 = np.divide(tt,y_exp)
#     print(num_train)
#     tt3 = np.power(tt2,1/num_train)
#     loss = np.log(np.prod(tt3))



    (C, D) = self.W.shape
    N = X.shape[0]

    scores = np.dot(self.W, X.T)
    scores -= np.max(scores) # shift by log C to avoid numerical instability

    y_mat = np.zeros(shape = (C, N))
    y_mat[y, range(N)] = 1

    # matrix of all zeros except for a single wx + log C value in each column that corresponds
    # quantity we need to subtract from each row of scores
    correct_wx = np.multiply(y_mat, scores)

    # create a single row of the correct wx_y + log C values for each data point
    sums_wy = np.sum(correct_wx, axis=0) # sum over each column

    exp_scores = np.exp(scores)
    sums_exp = np.sum(exp_scores, axis=0) # sum over each column
    result = np.log(sums_exp)

    result -= sums_wy

    loss = np.sum(result)
    loss /= num_train


    # vectorized gradient calculation #
    exp_a_sum = np.sum(exp_a,axis=0)

    y_mat_corres = np.zeros(shape = (num_classes, num_train))
    y_mat_corres[y, range(num_train)] = 1
    sum_exp_scores = np.sum(exp_a, axis=0)
    sum_exp_scores = 1.0 / exp_a_sum   # division by sum over columns
    exp_a *= sum_exp_scores
    grad = np.dot(exp_a, X)
    grad -= np.dot(y_mat_corres, X)
    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.
        # ================================================================ #
        mask = np.random.choice(num_train, batch_size, replace=True)

        X_batch = X[mask]                              # (dim, batch_size)
        y_batch = y[mask]                              # (batch_size,)

        pass


        # ================================================================ #
        # 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
        # ================================================================ #
        pass

        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.exp(self.W.dot(X.T)), axis=0)

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

return y_pred
```