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


    # print(X.shape)
```

```
    # print(self.W.T.shape)
    ex_sc = 0

    for i in range (X.shape[0]):
      e_x = np.exp(np.matmul(self.W, X[i])) # exp(wT * X) <-- exp of
score of each class

      ex_sc = e_x # sum of softmax scores, for the denominator

      ex_sc = ex_sc/(np.sum(ex_sc)) #

      log_sc = np.log(ex_sc) # take the log likelihood

      loss = loss - log_sc[y[i]]

    loss = loss/(X.shape[0])

    # ================================================================
#
    # 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)

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the
gradient
    #   as the variable grad.
    # ================================================================
#
    # ex_sc = 0
    # grad_sum = 0
    # smax_all = [X.shape[0]][self.W.shape[0]] # n x c
    # for i in range (X.shape[0]):
    #   e_x = np.exp(np.matmul(self.W, X[i])) # exp(wT * X) <-- exp of
```

```
score of each class
    #   ex_sc = e_x/(np.sum(e_x)) # normalize the scores - becomes row
of softmax
    #   smax_all[i] = ex_sc

    #   ex_sc += e_x # sum of softmax scores, for the denominator

    # print("x shape", X.shape) # 500
    # print("w shape", self.W.shape) # 10

    for i in range (X.shape[0]): # i is range 500
      e_x = np.exp(np.matmul(self.W, X[i])) # exp(wT * X) <-- exp of
score of each class (numerator)
      # print("e_x shape", e_x.shape[0]) # 500
      ex_sum = e_x/np.sum(e_x, axis=0) # ex/sum ex (so this adds in
the denominator)
      # print(np.shape(ex_sum))
      log_sc = np.log(ex_sum) # take the log likelihood

      loss = loss - log_sc[y[i]]

      for j in range (self.W.shape[0]): # j is range 20
        if (j == y[i]):
          grad[j] += (ex_sum[j]-1)*X[i]
        else:
          grad[j] += (ex_sum[j])*X[i]

    # print("ex sum", ex_sum)
    # print("loss before normalization", loss)
    # print("log sc", log_sc)
    grad = grad/(X.shape[0]) # normalize the grad
    loss = loss/(X.shape[0])
    print("for loop loss", loss)
    # print(grad.shape)



    # ================================================================
#
    # 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

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the softmax loss and gradient WITHOUT any for loops.
    # ================================================================
#
    e_x = np.exp(X.dot(self.W.T)) # exp(wT * X) <-- exp of score of
each class (numerator)
    ex_sum = e_x/np.sum(e_x, axis=1, keepdims=True) # ex/sum ex (so
this adds in the denominator) output is (10,)
    # print("ex_sum shape", ex_sum.shape)
    log_sc = np.log(ex_sum) # take the log likelihood

    log_sum = np.sum(log_sc[np.arange(X.shape[0]),y])
    loss = -log_sum
    loss = loss/(X.shape[0]) # normalize the loss
    # print("fast loss", loss)

    grad = (ex_sum).T.dot(X) # this is the default, otherwise if i ==
j then subtract X[i]
    # HOW TO DO MASKING?? for loss when i = j? use np arrange?
    mask_zero=np.zeros_like(ex_sum)
    mask_zero[np.arange(X.shape[0]),y]=1
    mask =  ex_sum-mask_zero # mask 500, 10
```

```
    # print("mask:", mask.shape)
    masked_grad = mask.T.dot(X) # X 500, 3072
    # grad -= masked_grad
    grad = masked_grad/(X.shape[0]) # normalize the grad  # 10, 3073
    # print("grad***", grad.shape)

    # ================================================================
#
    # 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: (batch_size, dim)
      #     - 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 = [batch_size][dim]
      # y_batch = [batch_size]

      # rand_list = np.random.choice(X.shape[0], batch_size)
      # for i in rand_list:
      #   X_batch.append(X[i])
      # rand_list = np.random.choice(y.shape[0], batch_size)
      # for i in rand_list:
      #   y_batch.append(y[i])

      b_samples = []
      b_labels = []

      for i in range(batch_size):
        index = np.random.choice(num_train)
        b_samples.append(X[index])
        b_labels.append(y[index])

      X_batch = np.array(b_samples)
      y_batch = np.array(b_labels)

      #
============================================================== #
      # 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 - grad * learning_rate

      #
=================================================================== #
      # 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[0])

    # ===================================================================
#
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ===================================================================
#
    # print("ypred shape", y_pred.shape)
    # X = X[:3073]
    p = (self.W @ X.T).T
    # print("pshape", p.shape)
    # print("xshape", X.shape)
    # print("wshape", self.W.shape)

    for i in range(y_pred.shape[0]):
      y_pred[i] = np.argmax(p[i])
    # ===================================================================
#
    # END YOUR CODE HERE
    # ===================================================================
```

```
#

    return y_pred
```