```python
import numpy as np
import matplotlib.pyplot as plt


class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, a hidden layer dimension of H, and performs classification over C classes.
  We train the network with a softmax loss function and L2 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

  The outputs of the second fully-connected layer are the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable self.params, which is a dictionary with the following keys:

    W1: First layer weights; has shape (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    """
    self.params = {}
    self.params['W1'] = std * np.random.randn(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_size)
    self.params['b2'] = np.zeros(output_size)


  def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optional; if it
      is not passed then we only return scores, and if it is passed then we
      instead return the loss and gradients.
    - reg: Regularization strength.

    Returns:
    If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those parameters
      with respect to the loss function; has the same keys as self.params.
    """
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass
    scores = None

    # ================================================================= #
    # YOUR CODE HERE:
    #   Calculate the output scores of the neural network.  The result
    #   should be (N, C). As stated in the description for this class,
    #   there should not be a ReLU layer after the second FC layer.
    #   The output of the second FC layer is the output scores. Do not
    #   use a for loop in your implementation.
    # ================================================================= #

    relu = lambda x: np.maximum(0, x)

    z1 = X @ W1.T + b1
    a1 = relu(z1)
    z2 =  a1 @ W2.T + b2

    scores = z2

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


    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None

    # ================================================================= #
    # YOUR CODE HERE:
    #   Calculate the loss of the neural network.  This includes the
    #   softmax loss and the L2 regularization for W1 and W2. Store the
    #   total loss in teh variable loss.  Multiply the regularization
    #   loss by 0.5 (in addition to the factor reg).
    # ================================================================= #

    # Softmax Loss Calc
    softmax_loss = np.sum(np.log(np.sum(np.exp(scores), axis = 1)) - scores[np.arange(N), y])
    # L2 regularization
    l2_reg = 0.5 * reg * (np.linalg.norm(W1)**2 + np.linalg.norm(W2)**2)
    # Final Loss
    loss = softmax_loss / N + l2_reg

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

    grads = {}

    # ================================================================= #
    # YOUR CODE HERE:
    #   Implement the backward pass.  Compute the derivatives of the
    #   weights and the biases.  Store the results in the grads
    #   dictionary.  e.g., grads['W1'] should store the gradient for
    #   W1, and be of the same size as W1.
    # ================================================================= #

    # Used denominator layout
    # Loss dL/dz2
    dL_dz2 = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)
    dL_dz2[np.arange(N), y] -= 1
    dL_dz2 = dL_dz2/N # normalize by samples
    # Loss dL/da1 = dz2/da1 * dL/dz2
    dL_da1 = (W2.T @ dL_dz2.T).T
    # Loss dL/dz1 = da1/dz1 * dL/da1
    dL_dz1 = (z1 > 0) * dL_da1

    grads['W2'] = dL_dz2.T @ a1 + (reg * W2)     # dL/dW2
    grads['b2'] = np.sum(dL_dz2,axis=0)                 # dL/db2

    grads['W1'] = (X.T @ dL_dz1).T + (reg * W1)      # dL/dW1
    grads['b1'] = np.sum(dL_dz1,axis=0)                 # dL/db1

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

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

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

      # ================================================================= #
      # YOUR CODE HERE:
      #   Create a minibatch by sampling batch_size samples randomly.
      # ================================================================= #

      samples = np.random.choice(X.shape[0], batch_size)
      X_batch = X[samples,:]
      y_batch = y[samples]

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

       # Compute loss and gradients using the current minibatch
      loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
      loss_history.append(loss)

      # ================================================================= #
      # YOUR CODE HERE:
      #   Perform a gradient descent step using the minibatch to update
      #   all parameters (i.e., W1, W2, b1, and b2).
      # ================================================================= #

      # Gradient descent step
      self.params['W1'] -= learning_rate * grads['W1']
      self.params['W2'] -= learning_rate * grads['W2']
      self.params['b1'] -= learning_rate * grads['b1']
      self.params['b2'] -= learning_rate * grads['b2']

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

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

      # Every epoch, check train and val accuracy and decay learning rate.
      if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch).mean()
        val_acc = (self.predict(X_val) == y_val).mean()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        learning_rate *= learning_rate_decay

    return {
      'loss_history': loss_history,
      'train_acc_history': train_acc_history,
      'val_acc_history': val_acc_history,
    }

  def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
      to have class c, where 0 <= c < C.
    """
    y_pred = None

    # ================================================================= #
    # YOUR CODE HERE:
    #   Predict the class given the input data.
    # ================================================================= #

    # Forward prop and argmax of scores
    z1 = X @ self.params['W1'].T + self.params['b1']
    a1 = np.maximum(0,z1)
    z2 =  a1 @ self.params['W2'].T + self.params['b2']

    y_pred = np.argmax(z2, axis=1)

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

    return y_pred
```