```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
  D, 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']
```

```python
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.
# ================================================================ #
softmax = lambda x: np.exp(x) / np.sum(np.exp(x), axis = 1, keepdims = True)
relu = lambda x: x * (x > 0)

h1 = relu(np.dot(X,W1.T) + b1)

scores = np.dot(h1,W2.T) + b2

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

# scores is num_examples by num_classes
probabilities = softmax(scores)
y_hat = probabilities[np.arange(N), y]
L2_reg_term =  0.5 * reg* (np.sum(W1**2) + np.sum(W2**2))
loss = np.sum(-np.log(y_hat)) / N + L2_reg_term

# ================================================================ #
# 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.
# ================================================================ #
softmax_grad = probabilities
softmax_grad[np.arange(N),y] -= 1
softmax_grad /= N

grads['W2'] = np.dot(softmax_grad.T,h1)
grads['b2'] = np.sum(softmax_grad, axis = 0)
```

```python
        grad_h2 = np.dot(softmax_grad,W2)

        da = grad_h2
        da[h1 <= 0] = 0 # relu
        grads['W1'] = np.dot(da.T,X)
        grads['b1'] = np.sum(da, axis = 0)

        grads['W1'] += reg * W1
        grads['W2'] += reg * W2

        # ================================================================ #
        # 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.
          # ================================================================ #
          minibatch_idx = np.random.choice(np.arange(X.shape[0]), size=batch_size, replace=True)
          X_batch = X[minibatch_idx]
          y_batch = y[minibatch_idx]
          # ================================================================ #
          # 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
```

```python
      #   all parameters (i.e., W1, W2, b1, and b2).
      # ================================================================ #

      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.
    # ================================================================ #
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    #softmax = lambda x: (np.exp(x)-np.max(x)) / np.sum((np.exp(x)- np.max(x)), axis = 1,
keepdims = True)
    softmax = lambda x: np.exp(x) / np.sum(np.exp(x), axis = 1, keepdims = True)
    relu = lambda x: x * (x > 0)

    h1 = relu(np.dot(X,W1.T) + b1)
    scores = np.dot(h1,W2.T) + b2

    probabilities = softmax(scores)
    y_pred = np.argmax(probabilities,axis = 1)
```

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

    return y_pred
```