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

def init_two_layer_model(input_size, hidden_size, output_size):
  """
  Initialize the weights and biases for 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. Weights are initialized to small
  random values and biases are initialized to zero.

  Inputs:
  - input_size: The dimension D of the input data
  - hidden_size: The number of neurons H in the hidden layer
  - ouput_size: The number of classes C

  Returns:
  A dictionary mapping parameter names to arrays of parameter values. It has
  the following keys:
  - W1: First layer weights; has shape (D, H)
  - b1: First layer biases; has shape (H,)
  - W2: Second layer weights; has shape (H, C)
  - b2: Second layer biases; has shape (C,)
  """
  # initialize a model
  model = {}
  model['W1'] = 0.00001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.00001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model

def two_layer_net(X, model, y=None, reg=0.0):
  """
  Compute the loss and gradients for 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 use a softmax loss function and L2
  regularization the the weight matrices. The two layer net should use a ReLU
  nonlinearity after the first affine layer.

  The two layer net 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.

  Inputs:
  - X: Input data of shape (N, D). Each X[i] is a training sample.
  - model: Dictionary mapping parameter names to arrays of parameter values.
    It should contain the following:
    - W1: First layer weights; has shape (D, H)
    - b1: First layer biases; has shape (H,)
    - W2: Second layer weights; has shape (H, C)
    - b2: Second layer biases; has shape (C,)
  - 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 not is passed, 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 passed, instead return a tuple of:
  - loss: Loss (data loss and regularization loss) for this batch of training
    samples.
```

```python
    - grads: Dictionary mapping parameter names to gradients of those parameters
      with respect to the loss function. This should have the same keys as model.
    """

    # unpack variables from the model dictionary
    W1,b1,W2,b2 = model['W1'], model['b1'], model['W2'], model['b2']
    N, D = X.shape

    # compute the forward pass
    scores = None
    #############################################################################
    # TODO: Perform the forward pass, computing the class scores for the input. #
    # Store the result in the scores variable, which should be an array of      #
    # shape (N, C).                                                             #
    #############################################################################

    # evaluate class scores with a 2-layer Neural Network
    hidden_layer = np.maximum(0, np.dot(X, W1) + b1) # note, ReLU activation
    scores = np.dot(hidden_layer, W2) + b2

    #############################################################################
    #                              END OF YOUR CODE                             #
    #############################################################################

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

    # compute the loss
    loss = None
    #############################################################################
    # TODO: Finish the forward pass, and compute the loss. This should include  #
    # both the data loss and L2 regularization for W1 and W2. Store the result  #
    # in the variable loss, which should be a scalar. Use the Softmax           #
    # classifier loss. So that your results match ours, multiply the           #
    # regularization loss by 0.5                                                #
    #############################################################################

    num_examples = X.shape[0]
    # get unnormalized probabilities
    exp_scores = np.exp(scores)
    # normalize them for each example
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    corect_logprobs = -np.log(probs[range(num_examples),y])
    # compute the loss: average cross-entropy loss and regularization
    data_loss = np.sum(corect_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W1*W1) + 0.5*reg*np.sum(W2*W2)
    loss = data_loss + reg_loss

    #############################################################################
    #                              END OF YOUR CODE                             #
    #############################################################################

    # compute the gradients
    grads = {}
    #############################################################################
    # TODO: Compute the backward pass, computing the derivatives of the weights #
    # and biases. Store the results in the grads dictionary. For example,       #
    # grads['W1'] should store the gradient on W1, and be a matrix of same size #
    #############################################################################

    # compute the gradient on scores
    dscores = probs
    dscores[range(N),y] -= 1
    dscores /= N

    # W2 and b2
```

```
    grads['W2'] = np.dot(hidden_layer.T, dscores)
    grads['b2'] = np.sum(dscores, axis=0)
    # next backprop into hidden layer
    dhidden = np.dot(dscores, W2.T)
    # backprop the ReLU non-linearity
    dhidden[hidden_layer <= 0] = 0
    # finally into W,b
    grads['W1'] = np.dot(X.T, dhidden)
    grads['b1'] = np.sum(dhidden, axis=0)

    # add regularization gradient contribution
    grads['W2'] += reg * W2
    grads['W1'] += reg * W1

    #############################################################################
    #                              END OF YOUR CODE                             #
    #############################################################################

    return loss, grads
```