```
import pandas as pd
import numpy as np
```

```
df  = pd.DataFrame([[8, 8, 4], [7, 9, 5], [6, 10, 6], [5, 12, 7]], columns = ['CGPA', 'ATS Score', 'Package(LPA)'])
df
```

	CGPA	ATS Score	Package(LPA)
0	8	8	4
1	7	9	5
2	6	10	6
3	5	12	7

Next steps: Generate code with df   View recommended plots   New interactive sheet

```
def initialize_parameters(layer_dimensions):
    # Set a fixed random seed to make the results reproducible
    np.random.seed(3)

    # Create an empty dictionary to store the weight and bias parameters
    parameters = {}

    # Get the number of layers in the network (including input and output layers)
    L = len(layer_dimensions)

    # Loop through layers 1 to L-1 (we skip layer 0 because it's the input layer)
    for l in range(1, L):
        # Initialize weights for layer l with small values (here all 0.1 for simplicity)
        # Shape: (number of neurons in previous layer, number of neurons in current layer)
        parameters['W' + str(l)] = np.ones((layer_dimensions[l - 1], layer_dimensions[l])) * 0.1

        # Initialize biases for layer l with zeros
        # Shape: (number of neurons in current layer, 1)
        parameters['b' + str(l)] = np.zeros((layer_dimensions[l], 1))

    # Return the dictionary containing all the initialized parameters
    return parameters
```

```
def linear_forward(A_prev, W, b):
    # Compute the linear part of a layer's forward propagation
    # A_prev: activations from the previous layer (shape: size of previous layer x number of examples)
    # W: weights matrix for the current layer (shape: size of previous layer x size of current layer)
    # b: bias vector for the current layer (shape: size of current layer x 1)

    # Compute the linear transformation Z = W^T * A_prev + b
    # Note: W.T is used because we want the shape to be (size of current layer x number of examples)
    Z = np.dot(W.T, A_prev) + b

    # Return the result of the linear transformation (Z)
    return Z
```

```
# Main Forward Propagation function for a single row instance (i.e., one data sample)
def L_layer_forward(X, parameters):
    # X: input vector of shape (input size, 1) - i.e., a single column (one data instance)
    # parameters: dictionary containing all the weights and biases for each layer

    A = X  # Initial activation is the input feature vector
    L = len(parameters) // 2  # Number of layers in the network (each has W and b)

    # Loop through all the layers in the network
    for l in range(1, L + 1):
        A_prev = A  # Store previous activation for computing current layer's activation

        # Retrieve current layer's weights and bias
        Wl = parameters['W' + str(l)]
        bl = parameters['b' + str(l)]

        # Perform linear forward computation: Z = W.T @ A_prev + b
        # NOTE: Your 'W' is stored as shape (prev_layer, current_layer), so we need W.T
        A = linear_forward(A_prev, Wl, bl)

        # Optional debugging prints:
        print("A" + str(l-1) + ": ", A_prev)
        print("W" + str(l) + ": ", Wl)
        print("b" + str(l) + ": ", bl)
        print("A" + str(l) + ": ", A)
        print("--" * 20)

    # Return final output (last activation), and the last activation before it (optional)
    return A, A_prev
```

```
# Extract the first row's input features ('CGPA' and 'ATS Score') from the DataFrame
# .values[0] gives the first row as a NumPy array → shape will be (2,)
```

```
# .reshape(2, 1) converts it to a column vector of shape (2,1), which is required for our neural network
X = df[['CGPA', 'ATS Score']].values[0].reshape(2, 1)

# Extract the actual output (label) corresponding to the first row: 'Package(LPA)'
# .values[0][0] gives the scalar value from the first row, first column
y = df[['Package(LPA)']].values[0][0]

# Initialize neural network parameters (weights and biases) for a 3-layer network:
# - 2 input neurons (for CGPA and ATS Score)
# - 2 hidden neurons in the first hidden layer
# - 1 output neuron (for predicting Package)
# All weights are initialized to 0.1, and all biases to 0
parameters = initialize_parameters([2, 2, 1])

# Print the initialized parameters to verify their structure
parameters

# Perform a full forward pass through the network using the input X and the initialized parameters
# This will compute activations at each layer and finally give the output prediction
y_hat,A1 = L_layer_forward(X, parameters)
```

```
A0:  [[8]
 [8]]
W1:  [[0.1 0.1]
 [0.1 0.1]]
b1:  [[0.]
 [0.]]
A1:  [[1.6]
 [1.6]]
----------------------------------------
A1:  [[1.6]
 [1.6]]
W2:  [[0.1]
 [0.1]]
b2:  [[0.]]
A2:  [[0.32]]
----------------------------------------
```

```
# Above:
# A0 = [8, 8] — This is the input vector (features from the first row of the dataset)
# W1 = Weights for Layer 1 — These connect the input layer to the first hidden layer
# b1 = [b11, b12] — Biases for the neurons in hidden layer 1

# These are passed to linear_forward(A_prev, W1, b1) which computes:
# Z1 = W1.T @ A0 + b1
# A1 = Z1 in this case (since there's no activation function yet), so:
# A1 = [O11, O12] — outputs from the first hidden layer (linear combinations)

# Next layer:
# A1 is now passed as input to the next layer with W2, b2 to get:
# A2 = [O21] — the output from the final layer (the prediction)

# The function L_layer_forward returns:
# - A2: the final output (predicted Package)
# - A1: the activation from the previous layer (needed for backprop)

# WHY RETURN A_prev TOO?
# During backpropagation, we compute gradients like:
# ∂L/∂W = A_prev.T @ dZ
# That’s why we need the activations of the previous layer (A_prev or Oij values)
# They're essential for calculating the gradients during the backward pass.
```

```
# y_hat is the final output returned from the forward propagation:
# It's a 2D array with shape (1, 1), something like: [[value]]
# To convert it into a plain scalar value (float), we extract the single element
y_hat = y_hat[0][0]
# A1 is the output from the hidden layer (before applying any activation, in your current setup).
# This matrix contains the values computed after the first linear transformation:
# A1 = W1.T @ A0 + b1
# These are useful not only for forward pass inspection but also critical in backpropagation
# since gradients depend on the activations from previous layers.
A1
#This will output something like: array([[O11], [O12]])
```

```
array([[1.6],
       [1.6]])
```

```
def update_parameters(parameters, y, y_hat, A1, X):
    # Compute the gradient of Mean Squared Error (MSE) Loss w.r.t y_hat:
    # dL/dy_hat = 2 * (y - y_hat)
    # Learning rate is hardcoded as 0.001 here

    # ================= Layer 2 (Output Layer) Updates =================

    # W2[0][0] is the weight connecting A1[0] to the output neuron
    # Gradient: dL/dW2 = dL/dy_hat * d(y_hat)/dW2 = 2*(y - y_hat) * A1[i]
    parameters['W2'][0][0] += 0.001 * 2 * (y - y_hat) * A1[0][0]

    # W2[1][0] connects A1[1] to output
```

```
    parameters['W2'][1][0] += 0.001 * 2 * (y - y_hat) * A1[1][0]

    # Bias b2 — add gradient: dL/db2 = 2 * (y - y_hat)
    parameters['b2'][0][0] += 0.001 * 2 * (y - y_hat)

    # ================== Layer 1 (Hidden Layer) Updates ==================

    # The hidden layer gradients are approximated manually here:
    # W1[0][0] is the weight from X[0] to A1[0], which connects to output via W2[0][0]
    # So, its gradient flows back via W2[0][0] path
    parameters['W1'][0][0] += 0.001 * 2 * (y - y_hat) * parameters['W2'][0][0] * X[0][0]

    # W1[0][1] connects X[1] to A1[0]
    parameters['W1'][0][1] += 0.001 * 2 * (y - y_hat) * parameters['W2'][0][0] * X[1][0]

    # Bias for neuron A1[0]
    parameters['b1'][0][0] += 0.001 * 2 * (y - y_hat) * parameters['W2'][0][0]

    # W1[1][0] connects X[0] to A1[1], which connects to output via W2[1][0]
    parameters['W1'][1][0] += 0.001 * 2 * (y - y_hat) * parameters['W2'][1][0] * X[0][0]

    # W1[1][1] connects X[1] to A1[1]
    parameters['W1'][1][1] += 0.001 * 2 * (y - y_hat) * parameters['W2'][1][0] * X[1][0]

    # Bias for neuron A1[1]
    parameters['b1'][1][0] += 0.001 * 2 * (y - y_hat) * parameters['W2'][1][0]
```

```
# Call the parameter update function with the following:
# parameters: current weights and biases
# y: actual label (Package value for this instance)
# y_hat: predicted output from forward propagation
# A1: activations (outputs) from the hidden layer
# X: input feature vector (CGPA, ATS Score for this row)
update_parameters(parameters, y, y_hat, A1, X)
```

```
# After the update, print or inspect the updated parameters dictionary
# This dictionary contains the learned weights and biases for both layers:
#   - 'W1' and 'b1' are for the layer between input and hidden layer
#   - 'W2' and 'b2' are for the layer between hidden layer and output
# Since one step of gradient descent has been applied, their values should now be
# slightly different from their initial values (e.g., 0.1 or 0.0)
parameters
```

```
{'W1': array([[0.13249307, 0.13781791],
        [0.1325153 , 0.1378465 ]]),
 'b1': array([[0.00439444],
        [0.00439761]]),
 'W2': array([[0.15747535],
        [0.15783471]]),
 'b2': array([[0.03262821]])}
```

```
# Extract input features for the second training example (index 1)
X = df[['CGPA', 'ATS Score']].values[1].reshape(2, 1) # Shape(no of features, no. of training exaplme)
y = df[['Package(LPA)']].values[1][0]

y_hat,A1 = L_layer_forward(X,parameters)
y_hat = y_hat[0][0]

parameters
```

```
A0:  [[7]
 [9]]
W1:  [[0.13249307 0.13781791]
 [0.1325153  0.1378465 ]]
b1:  [[0.00439444]
 [0.00439761]]
A1:  [[2.12448363]
 [2.20974148]]
----------------------------------------
A1:  [[2.12448363]
 [2.20974148]]
W2:  [[0.15747535]
 [0.15783471]]
b2:  [[0.03262821]]
A2:  [[0.71595594]]
----------------------------------------
{'W1': array([[0.13249307, 0.13781791],
        [0.1325153 , 0.1378465 ]]),
 'b1': array([[0.00439444],
        [0.00439761]]),
 'W2': array([[0.15747535],
        [0.15783471]]),
 'b2': array([[0.03262821]])}
```

```
update_parameters(parameters,y,y_hat,A1,X)
parameters
```

```
{'W1': array([[0.14302965, 0.15136494],
        [0.14311725, 0.15147757]]),
 'b1': array([[0.00589966],
        [0.00591218]]),
```

```
 'W2': array([[0.17567812],
        [0.17676797]]),
 'b2': array([[0.0411963]])}
```

```
# Extract input features for the third training example (index 2)
X = df[['CGPA', 'ATS Score']].values[2].reshape(2, 1) # Shape(no of features, no. of training exaplme)
y = df[['Package(LPA)']].values[2][0]

y_hat,A1 = L_layer_forward(X,parameters)
y_hat = y_hat[0][0]
```

```
A0:  [[ 6]
 [10]]
W1:  [[0.15528401 0.17178888]
 [0.15552293 0.1721537 ]]
b1:  [[0.00794206]
 [0.00797979]]
A1:  [[2.49487538]
 [2.76025011]]
----------------------------------------
A1:  [[2.49487538]
 [2.76025011]]
W2:  [[0.19921008]
 [0.20166995]]
b2:  [[0.05144876]]
A2:  [[1.10511257]]
----------------------------------------
```

```
update_parameters(parameters,y,y_hat,A1,X)
parameters
```

```
{'W1': array([[0.16841999, 0.19368218],
        [0.168956  , 0.19454215]]),
 'b1': array([[0.01013139],
        [0.01021863]]),
 'W2': array([[0.22363435],
        [0.22869217]]),
 'b2': array([[0.06123854]])}
```

```
# Extract input features for the 4th training example (index 2)
X = df[['CGPA', 'ATS Score']].values[3].reshape(2, 1) # Shape(no of features, no. of training exaplme)
y = df[['Package(LPA)']].values[3][0]

y_hat,A1 = L_layer_forward(X,parameters)
y_hat = y_hat[0][0]
```

```
A0:  [[ 5]
 [12]]
W1:  [[0.16841999 0.19368218]
 [0.168956   0.19454215]]
b1:  [[0.01013139]
 [0.01021863]]
A1:  [[2.87970328]
 [3.31313536]]
----------------------------------------
A1:  [[2.87970328]
 [3.31313536]]
W2:  [[0.22363435]
 [0.22869217]]
b2:  [[0.06123854]]
A2:  [[1.46292723]]
----------------------------------------
```

```
update_parameters(parameters,y,y_hat,A1,X)
parameters
```

```
{'W1': array([[0.18256857, 0.22763878],
        [0.18365041, 0.22980874]]),
 'b1': array([[0.0129611 ],
        [0.01315752]]),
 'W2': array([[0.2555246 ],
        [0.26538232]]),
 'b2': array([[0.07231268]])}
```

```
# Till now, the whole dataset got trained for 1 time(1 epoch)
# Let's start with epoch implementation
```

```
# Initialize parameters before training
# Input layer: 2 neurons (CGPA, Profile Score)
# Hidden layer: 2 neurons
# Output layer: 1 neuron (Predicted LPA)
parameters = initialize_parameters([2, 2, 1])

# Set the number of epochs — how many times the entire dataset is processed
epochs = 5
```

```
# Outer loop: iterate through entire dataset multiple times (epochs)
for i in range(epochs):

    # Initialize list to track loss for the current epoch
```

```
    Loss = []

    # Inner loop: iterate through each training example (row in the DataFrame)
    for j in range(df.shape[0]):

        # Extract input vector X for j-th training sample
        # Reshape to (features, 1) => column vector
        X = df[['CGPA', 'ATS Score']].values[j].reshape(2, 1)

        # Extract actual output (LPA) for j-th training sample
        y = df[['Package(LPA)']].values[j][0]

        # Perform forward pass — get prediction and hidden layer output
        y_hat, A1 = L_layer_forward(X, parameters)

        # Flatten the prediction output (from 2D to scalar)
        y_hat = y_hat[0][0]

        # Backward pass — update weights and biases based on prediction error
        update_parameters(parameters, y, y_hat, A1, X)

        # Calculate squared error and add to Loss list
        Loss.append((y - y_hat) ** 2)

    # After processing all samples in this epoch, print the average loss
    print('Epoch -', i + 1, 'Loss -', np.array(Loss).mean())
```

```
----------------------------------------
A0:  [[ 6]
 [10]]
W1:  [[0.15658396 0.18420354]
 [0.15743714 0.18540648]]
b1:  [[0.00866124]
 [0.00878791]]
A1:  [[2.52253643]
 [2.96807397]]
----------------------------------------
A1:  [[2.52253643]
 [2.96807397]]
W2:  [[0.20681313]
 [0.21470095]]
b2:  [[0.05447498]]
A2:  [[1.21341694]]
----------------------------------------
A0:  [[ 5]
 [12]]
W1:  [[0.16985018 0.2063139 ]
 [0.17140141 0.20868027]]
b1:  [[0.01087227]
 [0.01111529]]
A1:  [[2.91694011]
 [3.54684797]]
----------------------------------------
A1:  [[2.91694011]
 [3.54684797]]
W2:  [[0.23096179]
 [0.24311482]]
b2:  [[0.06404815]]
A2:  [[1.60004115]]
----------------------------------------
Epoch - 2 Loss - 19.438253848220803
A0:  [[8]
 [8]]
W1:  [[0.18402315 0.24032904]
 [0.186598   0.24515208]]
b1:  [[0.01370687]
 [0.01415461]]
A1:  [[2.97867611]
 [3.89800358]]
----------------------------------------
A1:  [[2.97867611]
 [3.89800358]]
W2:  [[0.2624645 ]
 [0.28142048]]
b2:  [[0.07484807]]
A2:  [[1.95362286]]
----------------------------------------
A0:  [[7]
 [9]]
W1:  [[0.19301593 0.24932182]
 [0.19633463 0.25488871]]
b1:  [[0.01483097]
 [0.01537168]]
A1:  [[3.1329542 ]
 [4.05462284]]
----------------------------------------
```

```
# After all epochs, print the final learned parameters
# These include W1, b1, W2, b2 after being refined over 5 epochs
parameters
```

```
{'W1': array([[0.273603  , 0.3993222 ],
        [0.28787155, 0.42586102]]),
 'b1': array([[0.02885522],
        [0.03133223]]),
 'W2': array([[0.42574893],
        [0.50219328]]),
 'b2': array([[0.11841278]])}
```

```
Start coding or generate with AI.
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.