Project Title: Breast Cancer Prediction Using a Custom Neural Network (No External Libraries Outside of Python)

Objective:

The main objective of this project is to build a **custom artificial neural network (ANN) from scratch** to predict outcomes for a specific application—**in this case, breast cancer prediction is considered**. However, neural network implementation is **generalizable** and can be adapted for various machine learning tasks based on different applications.

This project is developed **without using external libraries** such as **TensorFlow, Keras, PyTorch, scikit-learn, pandas, or NumPy**. Instead, all computations, including **matrix operations, data loading, standardization, training, and evaluation**, are manually implemented using **pure Python**.

Working Procedure

The project involves the following key steps:

Part A: Define Function and Class with Method

1.Matrix Multiplication without Numpy

Instead of using libraries like numpy, matrix multiplication is implemented manually in Python to understand the core mechanism of how weights and inputs are combined in a neural network.

2. Custom Neural Network Implementation

The neural network is implemented from scratch, following these key steps:

2.1 Initialization

The network begins by setting up the **learning rate**, which controls how quickly the model updates its weight during training. The layers of the network are initialized, and the default **loss function** is set to **Mean Squared Error (MSE)**.

2.2 Input Layer

The **input layer** is defined by specifying the number of **input features** (input dimensionality). This defines the size of the input vector, which is necessary for configuring the subsequent layers of the neural network.

2.3 Weight Initialization

Weights are initialized using **Xavier initialization** (also called **Glorot initialization**), which helps to scale the weights properly to improve convergence during training. Biases are initialized to zero.

2.4 Adding Hidden Layers

Hidden layers are added to the network with the specified number of neurons and the chosen **activation function**. The weight matrix size for each hidden layer is determined based on the number of neurons in the previous layer, and the activation function introduces non-linearity to the layer.

2.5 Activation Functions

The activation function determines the non-linearity applied to the output of each layer. The common activation functions include:

- **ReLU**: Outputs the input directly if it's positive; otherwise, it returns zero.

- **Sigmoid**: Compresses the output to a range between 0 and 1, typically used for binary classification tasks.
- **Linear**: No transformation is applied, used primarily for regression tasks.

2.6 Forward Propagation

Forward propagation is the process of calculating the output of the network by passing the input through each layer. Each layer computes a **weighted sum** of its inputs, adds the **bias**, and applies the **activation function**. The output of one layer is passed as input to the next layer until the final output is computed.

2.7 Backpropagation

Backpropagation is used to update the weights and biases by computing the gradients of the **loss function** with respect to each weight. The gradients are computed using the **chain rule** of calculus, and weights are updated via **gradient descent** to minimize the loss function.

2.8 Loss Function

The **loss function** measures the difference between the true values (y_true) and the predicted values (y_pred). The network can support:

- **Mean Squared Error (MSE)**: Calculates the average squared difference between predicted and actual values.
- **Binary Cross-Entropy**: Measures the difference between predicted probabilities and the actual binary labels, commonly used for binary classification tasks.

2.9 Derivative of Activation Functions

The **derivative** of the activation function is used during backpropagation to compute the gradients for updating the weights. The derivatives for the most common activation functions are:

- **ReLU**: Derivative is 1 for positive inputs and 0 for non-positive inputs.
- **Sigmoid**: Derivative is σ(x)(1−σ(x))\sigma(x)(1 - \sigma(x))σ(x)(1−σ(x)), where σ(x)\sigma(x)σ(x) is the sigmoid function.
- **Linear**: Derivative is 1 since the linear function does not alter its input.

2.10 Training the Model

The model is trained by iterating through the dataset for a set number of **epochs**. During each epoch, the data is shuffled, and mini batches are created. For each mini-batch, **forward propagation** and **backpropagation** are performed, and the weights are updated using the computed gradients.

2.11 Prediction

After training, the model can be used to **predict** new data. The **predict ()** function performs **forward propagation** on the new input data and returns the predicted output.

2.12 Compiling the Model

The model is **compiled** by specifying the **loss function** to be used during training. The loss function can either be **Mean Squared Error (MSE)** for regression tasks or **binary cross-entropy** for binary classification tasks. This step ensures that the correct loss function is applied during optimization.

3. Define Customized Label Encoder Function

Custom function to encode categorical labels (like the target variable) into numerical values.

4. Data Standardization

Data standardization is implemented manually to ensure each feature is centered and scaled, improving model convergence during training.

5. Define Training and Testing Dataset Splitting Function

Function to split the dataset into training and testing sets for model evaluation.

6. Accuracy and Performance Metrics without Scikit-Learn

Accuracy, precision, recall, and F1 score are computed manually to have complete control over the evaluation process.

Part B: Data Loading, Preprocessing, and Model Execution

7. Load Dataset - Breast Cancer Wisconsin (Diagnostic)

The dataset is loaded using Python's built-in functions, ensuring data preparation is handled without relying on external libraries.

8. Call Custom Label Encoder to the Target Variable

The custom label encoder is applied to convert the target variable into numerical values.

9. Separate Input Features and Target Labels

The dataset is split into **input features** (X) and **target labels** (y).

10. Call Data Standardization

Standardization is applied to the input features to scale them to a standard range.

11. Initialize, Train Custom Artificial Neural Network (ANN)

The neural network is initialized and trained on the preprocessed data, using the custom neural network implementation.

12. Test Custom Artificial Neural Network (ANN) on New Data

After training, the model is tested on new, unseen data to evaluate its performance.

Achievements

While the code may be complex, especially for the ANN class definition, this approach provides an in-depth understanding of the underlying mathematical concepts behind neural networks. By avoiding high-level machine learning and deep learning libraries, this project provides a **step-by-step understanding of how ANN works internally**, from **forward propagation and backpropagation to gradient descent optimization**. Additionally, this approach **enhances programming skills** in the field of **machine learning and deep learning**, reinforcing fundamental concepts that are often abstracted away by external frameworks.

Dependences to Run the Code

There are **no external dependencies** required to run the code. It is designed to run directly on Jupyter notebooks or any other platform supporting .ipynb files, using only Python's built-in functionalities.