Neural Network Documentation

This document provides a detailed guide on how to use the two neural network classes you have provided: `NeuralNetwork` (a simple network with one hidden layer) and `NeuralNetwork` (a more advanced feedforward network with multiple hidden layers, which I will refer to as `Feedforward_Neural_Network` to differentiate it from the first one).

1. Simple Neural Network (`NeuralNetwork`)

This class implements a simple neural network with a single hidden layer. It is designed for straightforward tasks and uses standard backpropagation for training.

Class: `NeuralNetwork`

Constructor

```
def __init__(self, input_size, hidden_size, output_size, activation_name='sigmoid'):
```

- `input_size`: The number of neurons in the input layer.
- `hidden_size`: The number of neurons in the single hidden layer.
- `output_size`: The number of neurons in the output layer.
- `activation_name`: (Optional) The activation function for the hidden layer. It can be `'sigmoid'` or `'Relu'`. Defaults to `'sigmoid'`.

Methods

- `train(x, y, learn_rate=0.01, epoch=2000, limite_error=10e-5, desplay_fr=1000)`:
    - Trains the neural network using the provided data.
    - `x`: Input data (e.g., a NumPy array).
    - `y`: Target data (e.g., a NumPy array).
    - `learn_rate`: (Optional) The learning rate for gradient descent. Defaults to `0.01`.
    - `epoch`: (Optional) The maximum number of training iterations. Defaults to `2000`.
    - `limite_error`: (Optional) The training will stop early if the mean squared error (MSE) falls below this limit. Defaults to `10e-5`.
    - `desplay_fr`: (Optional) The frequency (in epochs) to display the current loss. Defaults to `1000`.
- `predict(x)`:
    - Makes predictions on new data.
    - `x`: Input data for prediction.
    - Returns the predicted class labels as a NumPy array.

- predict_proba(x):
    - Calculates the probability distribution over classes.
    - x: Input data for prediction.
    - Returns the raw output from the output layer (e.g., probabilities if a softmax activation is used).

Example Usage (from NeuralNetwork.py)

```
if __name__ == "__main__":
    # Create an instance of the class
    nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=2)

    # XOR dataset
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])

    # One-hot encode the labels
    ohe = OneHotEncoder(sparse_output=False)
    y_encoded = ohe.fit_transform(y)

    # Train the network
    nn.train(X, y_encoded, epoch=20000, learn_rate=0.1)

    # Make predictions
    predictions = nn.predict(X)

    print("Predictions:", predictions)
    print("Original Labels:", y.flatten())
```

2. Feedforward Neural Network with Multiple Hidden Layers (`Feedforward_Neural_Network`)

This class, referred to as `nnm` in your `test FNN.ipynb`, is a more flexible implementation that can handle multiple hidden layers, various activation functions, and different optimization algorithms.

Class: `NeuralNetwork` (`Feedforward_Neural_Network`)

Constructor

```
def __init__(self, input_size, hidden_layers, output_size, activation_hidden='relu', activation_output='softmax', optimisation_algorithm_name="adam"):
```

- `input_size`: The number of neurons in the input layer.
- `hidden_layers`: A list of integers, where each integer represents the number of neurons in a hidden layer. For example, `[4, 16]` creates a network with two hidden layers of 4 and 16 neurons, respectively.
- `output_size`: The number of neurons in the output layer.
- `activation_hidden`: (Optional) The activation function for all hidden layers. Defaults to `'relu'`.
- `activation_output`: (Optional) The activation function for the output layer. Defaults to `'softmax'`.
- `optimisation_algorithm_name`: (Optional) The optimization algorithm to use. Defaults to `'adam'`.

Methods

- `train(x, y, epoch=1000, learn_rate=0.01, beta_1=0.9, beta_2=0.999, epsilon=1e-8, limite_error=10e-5, desplay_fr=100)`:
    - Trains the neural network. This method includes parameters for the Adam optimizer.
    - `x`: Input data.
    - `y`: Target data.
    - `epoch`: (Optional) The number of training epochs. Defaults to `1000`.
    - `learn_rate`: (Optional) The learning rate. Defaults to `0.01`.
    - `beta_1`, `beta_2`, `epsilon`: Parameters for the Adam optimizer. Defaults are provided.
    - `limite_error`: (Optional) The training will stop early if the categorical cross-entropy loss is below this limit. Defaults to `10e-5`.

    - `desplay_fr`: (Optional) The frequency (in epochs) to display the current loss. Defaults to `100`.
    - Returns `loss_history`, a list of loss values for each epoch.
- `predict(x)`:
    - Makes predictions on new data.
    - `x`: Input data for prediction.
    - Returns the predicted class labels and the raw output from the output layer.
- `predict_proba(x)`:
    - Calculates the probability distribution over classes.
    - `x`: Input data for prediction.
    - Returns the raw output from the output layer (probabilities).

Example Usage (from `test FNN.ipynb` and `Feedforward_Neural_Network.py`)

Here is a simplified example demonstrating how to create and train this network.

```
# Assuming you have loaded and preprocessed your data

# For a practical example, you can refer to `test FNN.ipynb`

# to see how to load and prepare the MNIST dataset.



# For this simplified example, let's use the XOR dataset again.

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([[0], [1], [1], [0]])



# One-hot encode the labels

ohe = OneHotEncoder(sparse_output=False)

y_encoded = ohe.fit_transform(y)



# Create a network with two hidden layers

# Layer 1 has 4 neurons, Layer 2 has 16 neurons.

nnm = NeuralNetwork(input_size=2, hidden_layers=[4, 16], output_size=2)
```

```
# Train the network

loss_history = nnm.train(X, y_encoded, epoch=20000, learn_rate=0.1)


# Make predictions

predictions, _ = nnm.predict(X)


print("Predictions:", predictions)

print("Original Labels:", y.flatten())
```

3. Notebooks for Testing and Usage

The files `test nn.ipynb` and `test FNN.ipynb` are Jupyter notebooks designed to demonstrate and test the functionality of the `NeuralNetwork` and `Feedforward_Neural_Network` classes, respectively.

3.1. `test nn.ipynb`

This notebook serves as a testing environment for the simple `NeuralNetwork` class. It is likely used to:

- Import the `NeuralNetwork` class and other necessary libraries like NumPy, TensorFlow, and Matplotlib.
- Load a dataset (such as the XOR dataset or a more complex one).
- Perform data preprocessing steps, which may include scaling or principal component analysis (`PCA`) as suggested by the imports.
- Instantiate and train the `NeuralNetwork` model.
- Evaluate the trained model's performance using metrics from `sklearn`, such as `accuracy_score`, `confusion_matrix`, and `classification_report`, and visualize the results using `seaborn`.

3.2. `test FNN.ipynb`

This notebook is a testbed for the more advanced `Feedforward_Neural_Network` class (referred to as `nnm` within the notebook). Its purpose is to showcase the extended capabilities of this class, which include:

- Importing the `Feedforward_Neural_Network` class and other required libraries.

- Handling more complex datasets, potentially for image classification, as indicated by the `PIL` and `tensorflow` imports.
- Training a multi-layered network with specified hidden layer sizes and advanced optimizers like `adam`.
- Visualizing the training process, specifically by plotting the loss history over epochs using the `loss_history_plot` function.
- Evaluating the model's performance on the dataset.