Neural Network and MNIST Classification Documentation

This document provides an overview of the Python code provided in the files `NeuralNetwork_mulip_hidden_layers.py` and `test nnm.ipynb`. The code implements a simple feedforward neural network and uses it to classify handwritten digits from the MNIST dataset.

`NeuralNetwork_mulip_hidden_layers.py`

This file defines a `NeuralNetwork` class, a core component for building and training a neural network with multiple hidden layers.

`class NeuralNetwork`

The `NeuralNetwork` class provides the structure and methods for a multi-layer perceptron.

`__init__(self, input_size, hidden_layers, output_size, activation_hidden='relu', activation_output='softmax')`

This is the constructor for the `NeuralNetwork` class. It initializes the network's architecture, including weights and biases for each layer.

- `input_size`: The number of features in the input data.
- `hidden_layers`: A list of integers, where each integer represents the number of neurons in a hidden layer.
- `output_size`: The number of neurons in the output layer, corresponding to the number of classes.
- `activation_hidden`: The activation function to use for the hidden layers. Defaults to `'relu'`.
- `activation_output`: The activation function to use for the output layer. Defaults to `'softmax'`.

Activation Functions

The class includes implementations of common activation functions:

- `sigmoid(self, x)`: The sigmoid function, which squashes values between 0 and 1.
- `Relu(self, x)`: The Rectified Linear Unit function, which returns the input if positive and 0 otherwise.
- `softmax(self, x)`: The softmax function, used for multi-class classification to convert logits into probabilities.

`forward(self, x)`

This method performs the forward pass of the neural network. It takes an input `x` and propagates it through the layers, calculating the activations and weighted sums at each step.

`backward(self, y, activations, weighted_sums, learning_rate)`

This method implements the backpropagation algorithm. It calculates the gradients of the loss with respect to the weights and biases and updates them to minimize the loss.

- `y`: The true labels for the input data.
- `activations`: The activations from the forward pass.
- `weighted_sums`: The weighted sums from the forward pass.
- `learning_rate`: The step size for updating the weights and biases.

`categorical_cross_entropy(self, y_true, y_pred)`

This is the loss function used for multi-class classification. It measures the performance of a classification model whose output is a probability value between 0 and 1.

`train(self, x, y, epochs, learning_rate, limite_error, desplay_fr)`

This method trains the neural network on the given data for a specified number of epochs.

- `x`: The training input data.
- `y`: The training labels.
- `epochs`: The number of training iterations.
- `learning_rate`: The step size for weight updates.
- `limite_error`: A threshold to stop training early if the loss falls below this value.
- `desplay_fr`: The frequency to display the training loss.

`predict(self, x)`

This method makes predictions on new data. It returns the predicted class labels and the probability distribution over the classes.

`predict_proba(self, x)`

This method returns the probability of each class for a given input.

`test nnm.ipynb`

This Jupyter notebook demonstrates how to use the `NeuralNetwork` class to classify MNIST digits.

1. Data Loading and Preprocessing

- The `load_data()` function uses `tensorflow.keras.datasets.mnist.load_data()` to load the MNIST dataset.
- The data is then reshaped and normalized to fit the neural network's input requirements.
- `StandardScaler` from `sklearn.preprocessing` is used to standardize the input features.

- `PCA` from `sklearn.decomposition` is applied for dimensionality reduction.
- `OneHotEncoder` is used to convert the integer labels into a one-hot encoded format for training.

2. Model Training

- An instance of the `NeuralNetwork` class is created.
- The `train()` method is called to train the model on the preprocessed training data. The `loss_history` is stored to visualize the training progress.

3. Model Evaluation

- The `test_model()` function is used to evaluate the trained model on the test dataset.
- It uses the `predict()` method to get predictions on the test data.
- It then prints a `classification_report` and a `confusion_matrix` from `sklearn.metrics` to assess the model's performance.
- A heatmap visualization of the confusion matrix is generated using `seaborn`.
- A plot of the training loss over epochs is displayed using `matplotlib`.

This documentation provides a clear understanding of the provided code, explaining the purpose and function of each class and method. The Jupyter notebook serves as a practical example of how to utilize the neural network for a real-world machine learning task.