Report: Using an MLP and a CNN to Learn the MNIST Dataset

This report details the key parts of the provided files for using a Multi-Layer Perceptron (**MLP**) to learn the MNIST dataset. The files include the implementation of the **MLP**, the necessary layers and nodes, and the script to train and evaluate the model.

1. File: `EDF_Percpetron.py`

This file contains the core components of the neural network, including the base `Node` class, various types of nodes (e.g., `Input`, `Parameter`, `Linear`, `ReLU`, `Softmax`, `Cross_Entropy`), and the `Conv` and `MaxPooling` layers.

- **Node Class**: The base class for all nodes in the network, providing the structure for forward and backward passes.
- **Input Node**: Represents the input data.
- **Parameter Node**: Represents trainable parameters (weights and biases).
- **Linear Node**: Implements a linear transformation (fully connected layer).
- **ReLU Node**: Implements the ReLU activation function.
- **Softmax Node**: Implements the softmax activation function for classification.
- **Cross_Entropy Node**: Implements the cross-entropy loss function for multi-class classification.
- **Conv and MaxPooling Nodes**: Implement convolutional and max-pooling operations for **CNN**s.

2. File: `Nueron_Layer.py`

This file defines the layers of the neural network, including input layers, computation layers, and specific types of computation layers (e.g., `Linear_Computation_Layer`, `Linear_Softmax_Computation_Layer`, `Conv_layer`).

- **Nueron_Layer Class**: The base class for all layers, providing the structure for forward and backward passes.
- **Input_Layer Class**: Represents the input layer of the network.
- **Computation_Layer Class**: Represents a layer with an operation and activation function, as well as trainable parameters.
- **Linear_Computation_Layer Class**: A computation layer with a linear transformation and an activation function.
- **Linear_Softmax_Computation_Layer Class**: A computation layer with a linear transformation followed by a softmax activation function.
- **Conv_layer Class**: A convolutional layer with optional max-pooling.

3. File: `**MLP**.py`

This file defines the `**MLP**` class, which combines all **MLP** functionality into one class to reduce code clutter. It includes methods for building the network, training, and evaluation.

- ****MLP** Class**:
  - **Initialization**: Initializes the **MLP** with the specified number of features, outputs, depth, width, hidden layer type, output layer type, loss node type, and test node.
  - **Learn Method**: Trains the **MLP** on the training data for a specified number of epochs, batch size, and learning rate.
  - **Evaluate Method**: Evaluates the **MLP** on the test data and calculates accuracy and average entropy.
  - **Forward Pass Method**: Performs a forward pass through the network.
  - **Backward Pass Method**: Performs a backward pass through the network.
  - **SGD Update Method**: Updates the trainable parameters using stochastic gradient descent.

4. File: `full_MNIST.py`

This file contains the script to load the MNIST dataset, preprocess the data, define the **MLP** architecture, train the model, and evaluate its performance.

- **Loading the Dataset**: Loads the MNIST dataset using Keras and reshapes the data to fit the **MLP** model.
- **Defining Constants**: Defines constants such as the number of features, number of outputs, learning rate, number of epochs, batch size, depth, and width of the **MLP**.
- **Evaluation Function**: Defines a function to evaluate the model's predictions against the true labels.
- **Hot-One Encoding**: Converts the labels to one-hot encoded vectors.
- ****MLP** Initialization**: Initializes the** MLP** with the specified architecture.
- **Training the** MLP****: Trains the **MLP** on the training data and logs the loss and learning time.
- **Evaluating the** MLP****: Evaluates the **MLP** on the test data and logs the accuracy and average entropy.

5. Observations from `full_MNIST_evaluations.txt`

The accuracy of the model varies with different hyperparameters such as depth, width, learning rate, batch size, and epochs. Here are some observations:

- **Depth and Width**:

  - Increasing the depth generally improves accuracy. For example, with `Depth: 3` and `Width: 64`, the accuracy reaches up to 99.50%.
  - Lower depths tend to have lower accuracy. For example, with `Depth: 1` and `Width: 64`, the accuracy is around 83.45% to 97.65%.
- **Learning Rate**:
  - A moderate learning rate (e.g., 0.04) tends to yield better accuracy. For example, with `Depth: 1`, `Width: 64`, and `Learning rate: 0.04`, the accuracy is 93.13%.
  - Very high learning rates (e.g., 0.16) result in lower accuracy. For example, with `Depth: 1`, `Width: 64`, and `Learning rate: 0.16`, the accuracy is 88.48%.
- **Batch Size**:
  - Larger batch sizes (e.g., 300) with moderate learning rates tend to perform well. For example, with `Depth: 1`, `Width: 64`, `Learning rate: 0.04`, and `Batch size: 300`, the accuracy is 92.72%.
  - Smaller batch sizes (e.g., 100) with the same learning rate and depth can also perform well but may take longer to train.
- **Epochs**:
  - More epochs generally improve accuracy. For example, with `Depth: 3`, `Width: 64`, `Learning rate: 0.02`, `Batch size: 100`, and `Epochs: 1000`, the accuracy is 99.50%.
  - Fewer epochs result in lower accuracy. For example, with `Depth: 1`, `Width: 64`, `Learning rate: 0.08`, `Batch size: 150`, and `Epochs: 10`, the accuracy is 83.45%.

In summary, the accuracy improves with increased depth, moderate learning rates, larger batch sizes, and more epochs.

Summary

The provided files implement a Multi-Layer Perceptron (**MLP**) to learn the MNIST dataset. The key components include:

- **Core Nodes and Layers**: Defined in `EDF_Percpetron.py` and `Nueron_Layer.py`, these files provide the building blocks for the **MLP**.
- ****MLP** Class**: Defined in `**MLP**.py`, this class combines all** MLP** functionality and provides methods for training and evaluation.
- **Training Script**: Defined in `full_MNIST.py`, this script loads the dataset, preprocesses the data, defines the **MLP** architecture, trains the model, and evaluates its performance.

By following the structure and methods provided in these files, the **MLP** can be effectively trained to classify handwritten digits from the MNIST dataset.

6. A Convulotional Network

CNN Class

The `**CNN**` class encapsulates the functionality of a Convolutional Neural Network (**CNN**). Here's a detailed explanation of its components and methods:

Initialization (`__init__` method)

- **Attributes**:
  - `n_features`: Number of input features.
  - `n_outputs`: Number of output classes.
  - `architecture`: Architecture of the **CNN**, specifying the number of layers and their configurations.
  - `graph`: List to hold the layers of the network.
  - `input_layer`: The input layer of the network.
  - `output_layer`: The output layer of the network.
  - `test_node`: Node for holding test data.
  - `loss`: Loss function node.
  - `trainable`: List of trainable layers in the network.
- **Building the Graph**:
  - The input layer is created and added to the graph.
  - Convolutional layers are created based on the architecture and added to the graph.
  - A flattener node is added to flatten the output of the last convolutional layer.
  - A linear computation layer with ReLU activation is added.
  - The output layer is created using a linear softmax computation layer.
  - The loss function (cross-entropy) is added to the graph.
  - The trainable layers are identified and stored.

Methods

- `learn`: Trains the network using the provided training data.
  - Iterates over epochs and batches.
  - Performs forward and backward passes.
  - Updates the model parameters using stochastic gradient descent (SGD).
  - Tracks and prints the loss and training progress.
- `evaluate`: Evaluates the network on the test data.
  - Performs forward passes on test batches.
  - Computes the entropy and accuracy of the predictions.
- `forward_pass`: Executes the forward pass for all layers in the graph.
- `backward_pass`: Executes the backward pass for all layers in the graph.

- `sgd_update`: Updates the model parameters using SGD.

Conv Node

The `Conv` class represents a convolutional layer in the network. Here's a detailed explanation:

Initialization (`__init__` method)

- **Attributes**:
  - `inputs`: List of input nodes (parameters and input data).
  - `outputs`: List of output nodes.
  - `value`: Output value of the node.
  - `gradients`: Gradients of the node with respect to its inputs.

Methods

- `forward`: Performs the forward pass of the convolutional layer.
  - Pads the input data.
  - Computes the convolution operation by sliding the kernel over the input data.
  - Adds the bias to the result.
- `backward`: Performs the backward pass of the convolutional layer.
  - Computes the gradients of the loss with respect to the input, weights, and bias.
  - Uses the chain rule to propagate the gradients backward through the network.

MaxPooling Node

The `MaxPooling` class represents a max-pooling layer in the network. Here's a detailed explanation:

Initialization (`__init__` method)

- **Attributes**:
  - `inputs`: List of input nodes.
  - `outputs`: List of output nodes.
  - `value`: Output value of the node.
  - `gradients`: Gradients of the node with respect to its inputs.
  - `_cache`: Cache to store the max-pooling masks for the backward pass.

Methods

- `forward`: Performs the forward pass of the max-pooling layer.
  - Divides the input data into non-overlapping 2x2 regions.
  - Computes the maximum value in each region.
  - Stores the max-pooling masks in the cache for the backward pass.

- `backward`: Performs the backward pass of the max-pooling layer.
  - Uses the max-pooling masks to propagate the gradients backward through the network.
  - Sets the gradient of the previous layer to the gradient from the current layer.
- `_save_mask`: Saves the max-pooling mask for a given region.
  - Identifies the maximum value in each region and creates a mask.
  - Stores the mask in the cache for use in the backward pass.

Summary

- The `**CNN**` class combines all the convolutional functionality into one class, reducing code clutter and providing a structured way to build and train a **CNN**.
- The `Conv` node performs the convolution operation, which is essential for extracting features from the input data.
- The `MaxPooling` node performs max-pooling, which reduces the spatial dimensions of the input data and helps in making the network more robust to spatial variations.

7. Observations on Accuracy with Respect to Batch Size, Learning Rate, and Epochs

Batch Size

- **Low Batch Sizes (32, 64)**:
  - Accuracy is generally lower, ranging from 12.52% to 72.50%, depending on the learning rate and epochs. Moderate improvements are seen with increased epochs and appropriate learning rates.
- **Moderate Batch Sizes (128, 150)**:
  - Accuracy is generally higher compared to low batch sizes, achieving up to 97.56% with optimal learning rates and sufficient epochs. However, accuracy can drastically drop with inappropriate learning rates or fewer epochs.
- **High Batch Sizes (200, 256)**:
  - Shows good performance with higher learning rates and epochs, achieving up to 82.78% accuracy. Accuracy tends to be lower with fewer epochs or inappropriate learning rates.

Learning Rate

- **Low Learning Rates (e.g., 0.001, 1.5e-05)**:
  - Generally results in poor accuracy, indicating insufficient learning.
- **Moderate Learning Rates (e.g., 0.015, 0.02)**:
  - Accuracy improves significantly with appropriate batch sizes and epochs, reaching up to 72.50%.
- **High Learning Rates (e.g., 0.1, 0.2, 0.4, 0.8, 1)**:

  - Shows good improvement in accuracy, achieving up to 92.08% for specific combinations of batch size and epochs.
  - Very high learning rates (e.g., 8.5) can lead to poor accuracy (10.28%).
- **Very High Learning Rates (e.g., 3.2, 3.8, 4.5)**:
  - Can yield high accuracy (up to 97.56%) when combined with appropriate batch sizes and epochs.
  - However, can also lead to poor accuracy if not tuned properly.

Epochs

- **Few Epochs (e.g., 2, 4, 5)**:
  - Accuracy is generally lower, especially with lower learning rates and larger batch sizes.
- **Moderate Epochs (e.g., 10, 12, 14, 24)**:
  - Significant improvement in accuracy when combined with appropriate learning rates and batch sizes.
  - Longer training times but better accuracy.
- **Many Epochs (e.g., 64)**:
  - Continued improvement in accuracy but with diminishing returns, leading to longer learning times.

Summary

- **Optimal Combinations**:
  - Moderate batch sizes (e.g., 64, 150, 200) with moderate to high learning rates and sufficient epochs lead to the best accuracy.
  - Examples include batch size 150, learning rate 3.2, epochs 14 with 97.56% accuracy, and batch size 200, learning rate 1, epochs 5 with 82.78% accuracy.
- **Poor Combinations**:
  - Very low or very high learning rates with insufficient epochs or inappropriate batch sizes lead to poor accuracy.
  - Examples include batch size 64, learning rate 0.02, epochs 4 with 27.18% accuracy, and batch size 100, learning rate 8.5, epochs 5 with 10.28% accuracy.

8. Comparison of Evaluations

Here is a comparison of the evaluations from `full_mnist_evaluations.txt` and `full_mnist_conv_evaluations.txt` in terms of ideal hyperparameters, accuracy, learning time, and entropy:

`full_mnist_evaluations.txt`

- **Ideal Hyperparameters**:
  - **Depth**: 3
  - **Width**: 64

    - **Learning Rate**: 0.02
    - **Batch Size**: 100
    - **Epochs**: 1000
- **Best Accuracy**: 99.50%
    - **Loss**: 0.026350860164423465
    - **Learning Time**: 3265.9534 seconds
    - **Average Entropy**: 0.0525594503468861
- **Other Notable Results**:
    - **Accuracy**: 98.15%
        * **Loss**: 0.08344912031794338
        * **Learning Time**: 1474.9261 seconds
        * **Average Entropy**: 0.1389339241206964
    - **Accuracy**: 97.65%
        * **Loss**: 0.09889235988032277
        * **Learning Time**: 597.1605 seconds
        * **Average Entropy**: 0.1985149422127384

`full_mnist_conv_evaluations.txt`

- **Ideal Hyperparameters**:
    - **Architecture**: `[(1, 4), (1, 16), (1, 32), (1, 64), (1, 128)]`
    - **Learning Rate**: 3.2
    - **Batch Size**: 150
    - **Epochs**: 14
- **Best Accuracy**: 97.56%
    - **Loss**: 0.025931798373356893
    - **Learning Time**: 611.9142 seconds
    - **Average Entropy**: 0.09959452025238154
- **Other Notable Results**:
    - **Accuracy**: 97.07%
        * **Loss**: 0.08906416136021517
        * **Learning Time**: 930.7585 seconds
        * **Average Entropy**: 0.11518866117807991
    - **Accuracy**: 96.99%
        * **Loss**: 0.09692254804609869
        * **Learning Time**: 968.2003 seconds
        * **Average Entropy**: 0.14707633050950597

Summary

- **Ideal Hyperparameters**:
    - For **MLP**s, the ideal hyperparameters are a depth of 3, width of 64, learning rate of 0.02, batch size of 100, and 1000 epochs.
    - For **CNN**s, the ideal hyperparameters are an architecture of `[(1, 4), (1, 16), (1, 32), (1, 64), (1, 128)]`, learning rate of 3.2,

batch size of 150, and 14 epochs.

- **Accuracy**:
  - The highest accuracy in **MLP**s is 99.50%, while in **CNN**s it is 97.56%.
- **Learning Time**:
  - The learning time for the highest accuracy in **MLP**s is 3265.9534 seconds, while in **CNN**s it is 611.9142 seconds.
- **Entropy**:
  - The average entropy for the highest accuracy in **MLP**s is 0.0525594503468861, while in **CNN**s it is 0.09959452025238154.

In conclusion, while **MLP**s achieves a higher accuracy, it requires significantly more learning time. On the other hand, **CNN**s achieves slightly lower accuracy but with much less learning time, making it more efficient.