## Assignment 1 Part A

## Objectives:

1. Understand logistic regression and its implementation.
2. Familiarize with computational graph design and its forward/backward passes.
3. Explore batch processing and analyze its impact on model performance and efficiency.

---

**Tasks and Results:**

**Task 1: Design of Linear Node**

- **Objective:** Implement a linear node to perform the operation u=Wx+bu.
- **Implementation:**
  - Developed a `Linear` node capable of handling forward propagation Wx+b and backpropagation to compute gradients of weights W and biases b.
  - The node is flexible, allowing integration into various computational graphs.
- **Outcome:**
  - The `Linear` node performed efficiently, enabling accurate gradient updates during training. ○ Its modular design allowed seamless connection with subsequent components like activation and loss nodes.

---

**Task 2: Logistic Regression**

- **Objective:** Use the `Linear` node to implement logistic regression for binary classification.
- **Dataset:**
  - Two classes were drawn from Gaussian distributions to create a dataset simulating a linearly separable problem.
  - Features: Two-dimensional input (Feature 1, Feature 2).
- **Implementation:**
  - The computation graph included:
    - A `Linear` node for the Wx+b transformation.
    - A `Sigmoid` activation node for output probabilities. ▪ A `Binary Cross-Entropy (BCE)` loss node for error computation.
  - Forward and backward passes were implemented iteratively, leveraging batch processing for faster execution.
- **Outcome:**

- The model achieved an accuracy of **96%** on the test set, confirming the successful implementation. ○ The effectiveness of logistic regression in handling linearly separable data was demonstrated.

---

**Task 3: Batch Processing**

- **Objective:** Modify the training loop to operate on batches of varying sizes. **Challenges:**
  - Designing a flexible system that processes different batch sizes without introducing errors in gradient computation.
  - Ensuring the computational graph updates correctly with batch-wise inputs.
- **Implementation:**
  - The program processed batches of sizes 1,2,4,8,…, with the batch size doubling each iteration until constrained by system memory.
  - Loss and accuracy metrics were computed and recorded for each configuration. **Outcome:**
  - Batch processing significantly improved training efficiency by reducing computational overhead for larger datasets.
  - Generated training loss curves demonstrated the effect of batch size on loss convergence, illustrating a trade-off between stability and speed.

---

**Task 4: Effect of Batch Size on Training Loss**

- **Objective:** Analyze the impact of batch size on training loss convergence.
- **Approach:** ○ Plots of training loss for various batch sizes were generated to visualize trends. ○ Decision boundary plots were created to validate the model's performance visually.
- **Outcome:**
  - Larger batch sizes resulted in smoother convergence curves but required more memory and time per epoch.
  - Smaller batch sizes introduced noise in the convergence curve, sometimes leading to faster exploration of the loss landscape.

---

**Additional Insights:**

1. **Computational Graph Design:**
   - A clear separation of computation into nodes (e.g., input, linear, activation, and loss nodes) ensured modularity and scalability. ○ The forward and backward passes were implemented as reusable functions, simplifying the training loop.
2. **Batch Processing Benefits:**

    - Reduced computational redundancy by processing multiple examples simultaneously.
    - Improved gradient stability and convergence speed compared to sample-wise updates.
3. **Visualization:**
    - Decision boundary plots validated the model's capacity to generalize to unseen data.

## Assignment 1 Part B

### Objectives

This part of the assignment aimed to:

- Address the XOR problem using logistic regression and assess its limitations.
- Explore the effectiveness of multi-layer perceptrons (MLPs) in solving non-linear problems.
- Refactor and automate code for improved maintainability and efficiency.
- Develop a neural network for handwritten digit classification using the MNIST dataset.

---

**Task 1: XOR Problem**
A dataset was generated with two classes drawn from four Gaussian distributions. The means of the distributions were positioned to make the classes non-linearly separable, simulating an XOR problem.

- **Implementation:** A logistic regression model was trained on the generated dataset.
- **Performance:** The model's accuracy ranged between 46% and 50%, highlighting its inability to solve the XOR problem effectively due to the inherent linearity of logistic regression.
- **Insights:** This task reinforced the limitations of logistic regression for handling nonlinear problems, motivating the exploration of neural networks for such tasks.

---

**Task 2: Multi-Layer Perceptrons (MLPs)**
A multi-layer perceptron was constructed using the previously implemented `Linear` class to address the XOR problem.

- **Architecture:**
    - Two hidden layers, each containing 20 neurons.
    - Sigmoid activation function in all layers.
    - Softmax activation function at the output layer.

- **Implementation:** The MLP was trained on the XOR dataset using gradient descent.
- **Performance:** The accuracy ranged between 96% and 100%, effectively solving the XOR problem.
- **Insights:**
    - The importance of initializing weights correctly was highlighted during this task. Proper initialization significantly contributed to achieving high accuracy and stable convergence.
    - A parameter `initialization_controller` was introduced in the `Linear` module, which scales the random initial weights by a small factor to ensure they remain within a manageable range. This adjustment improved the training stability and underscored the critical role of proper initialization in training deep neural networks.

---

**Task 3: Code Refactoring and Automation (Updated)**

This task focused on improving the maintainability of the codebase by automating repetitive tasks and enhancing the usability of core components.

- **Improvements:**
    0. Automated the creation of parameter nodes (weights and biases) within the `Linear` class, reducing manual effort.
    1. Introduced a topological sorting algorithm to automate the creation of the computation graph and the trainable parameter list.
    2. Enhanced the `Linear` node to accept additional parameters, specifically: ▪ **Input Size:** Specifies the size of the input data.
        - **Output Size:** Defines the required output dimensions for the layer.
        - **Initialization Controller:** A scaling factor used to adjust the magnitude of initial weights. This parameter ensures weights are appropriately scaled, improving the stability and efficiency of the training process.
    3. Implemented a new method, `find_graph_and_trainable`, to streamline the process of defining the computation graph and identifying trainable parameters. This function:
        - Takes the final node of the network as input.
        - Utilizes a topological sorting algorithm to traverse the computation graph, ensuring nodes are processed in the correct order.
        - Identifies trainable nodes (parameters such as weights and biases) by checking the newly added `trainable` attribute of each node.
- **Benefits:**
  The refactored code is cleaner, more modular, and scalable, enabling faster experimentation with new architectures. The enhanced `Linear` node design reduced manual configuration and made it easier to build networks with varying input and output dimensions while maintaining effective initialization practices.

The `find_graph_and_trainable` method significantly simplified the process of defining the training workflow. By automating the identification of the computation graph and trainable parameters, the framework became more robust and adaptable to complex architectures. This feature also minimized potential errors caused by manual configuration, streamlining the development of neural networks.

---

**Task 4: Handwritten Digit Classification (MNIST Dataset)**
A neural network was developed to classify handwritten digits using a simplified version of the MNIST dataset.

- **Dataset:** The dataset contained 8x8 grayscale images, flattened into 64-dimensional vectors. A 60%-40% train-test split was used.
- **Architecture:**
  - One hidden layer with 64 neurons. ○ Softmax activation function at the output layer.
  - Cross-entropy loss function.
- **Implementation:**
  - Instead of creating a one-hot vector target, the target values were handled directly as scalar labels in the Softmax and cross-entropy computations. This simplification proved effective and avoided unnecessary computational overhead.
  - Both the Softmax activation and the cross-entropy loss were implemented in the custom library developed for this assignment. Designing the backward pass for these layers was particularly challenging but essential for achieving correct gradients during training.
- **Performance:** The model achieved an accuracy of 96%, meeting the assignment's requirements. **Insights:**
  - This task demonstrated the robustness of the implemented library and its ability to support multi-class classification with custom layers. ○ Despite the complexity of deriving the backward pass, the implementation was successful, paving the way for future enhancements and the application of the library to other classification problems.

## Assignment 1 Part C Report

### Objectives

The primary focus of this part was to extend the capabilities of the custom library developed in earlier parts of the assignment by:

- Utilizing the full MNIST dataset to train a Multi-Layer Perceptron (MLP) with optimized performance.

- Implementing convolution and max-pooling operations for image processing.
- Constructing a Convolutional Neural Network (CNN) to classify handwritten digits.

---

**Tasks and Results**

**Task 1: Full MNIST**

The full-resolution MNIST dataset was loaded using the provided `Full_MNIST.py` script. The goal was to train an MLP with optimal performance by experimenting with various hyperparameters.

- **Dataset**: The MNIST dataset consisted of 28x28 grayscale images, flattened into 784dimensional input vectors.
- **Experiments**:
    0. **Network Depth**: Two hidden layers (128 and 64 neurons) were used.
    1. **Activation Functions**: ReLU was chosen due to its faster convergence compared to Sigmoid.
    2. **Hyperparameter Tuning**: Learning rate (0.001) and batch size (128) were selected based on previous successful experiments.
- **Results**:
    - The MLP achieved a test accuracy of **97%-98%**, demonstrating its effectiveness for this dataset.

---

**Task 2: Convolution and Max Pooling**

Custom classes for convolution and max-pooling operations were implemented to process feature maps effectively.  **Conv Class**:

    - Designed to perform convolution operations using a specified kernel size and stride.
    - Included forward and backward passes for gradient computation. ○ **Challenges**: Implementing the backward pass required meticulous derivation of gradients for both the kernel and the input.
- **MaxPooling Class**:
    - Performed down sampling of feature maps using a pooling window and stride. ○ **Challenges**: Efficiently mapping gradients during the backward pass was nontrivial, particularly for non-overlapping pooling windows.
- **Flatten Node**:

- The **Flatten Node** was essential to bridge the gap between the output of the final max-pooling layer and the fully connected (linear) layers.
- By converting multi-dimensional feature maps into 1D vectors, it ensured compatibility with the linear layer's input requirements.

---

**Task 3: Constructing the CNN**

A CNN was built using the `Conv` and `MaxPooling` classes, following the specified architecture:

- **Architecture**:
  - Conv(1, 16), MaxPooling ○ Conv(16, 32), MaxPooling ○ Conv(32, 64), MaxPooling ○ Conv(64,128), MaxPooling ○ Flatten, Linear(128, 10)
- **Activation Function**: ReLU was applied after every convolution operation.
- **Loss and Optimization**: The Cross-Entropy loss and Stochastic Gradient Descent (SGD) were used for training.
- **Challenges and Results**:
  - Due to the long execution time of approximately one hour for two epochs, it was not feasible to extensively experiment with hyperparameters such as batch size and learning rate. ○ Consequently, the CNN did not surpass the MLP in terms of accuracy, remaining in the **93%-94% range**.
  - This underscores the importance of optimizing computational efficiency for training deeper models.

---

**Key Findings and Insights**

1. **MLP vs. CNN Performance**: Despite CNNs' ability to extract hierarchical features, the MLP achieved comparable accuracy due to limited optimization and time constraints during CNN training.
2. **Flatten Node**: The introduction of the flatten node played a crucial role in seamlessly connecting max-pooling outputs to fully connected layers. This node proved to be a vital component in the CNN architecture.
3. **Time and Resources**:
   - Training CNNs on large datasets requires significant computational resources and time.
   - Optimizing training efficiency is crucial for achieving the full potential of deeper architectures.
4. **Custom Library Challenges**: Implementing convolutional operations and max pooling highlighted the complexity of deriving gradients for non-linear and multi-dimensional transformations.