AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

MECHATRONICS ENGINEERING DEPARTMENT

CSE473 | Computational Intelligence - Fall 2025

Project Milestone(1) Report

Team Number (2)

Name	Section	ID
Marwan Mahmoud Ali	2	2100771
Mohamed Alaa Abdelkareem	2	2100392
Omar Khaled Ahmed	3	2100705
Omar Emad El Din Hassan	3	2100580
Adham Waleed Gamal	2	2100451

REPO: Neural-Network-Library_Advanced-Applications

Submitted to:
Dr. Hossam Hassan
Eng. Abdallah Awdallah

Contents

I. Library & Architecture Design

1. The Base Class Paradigm

The core design principle is the use of base classes for Layer, Activation, and Loss.

- Every component must implement a consistent interface, primarily the forward(x) and backward(grad) methods.
- This adherence to the interface allows the Model to chain any combination of these components together without knowing their internal details.

2. The Model Orchestrator (network.py)

The Model class acts as the central orchestrator, managing the entire lifecycle:

- Sequential Container: It holds an ordered list of layers (self.layers).
- Training Loop (fit): It coordinates the training process:
  1. Mini-Batching: Data is split into batches using to_batches.
  2. Forward Pass: Data flows through the layers via self._forward(x).
  3. Loss Calculation: The difference is measured using self._loss.forward().
  4. Backward Pass: The gradient starts at the loss and flows in reverse via self._backward(grad).
  5. Parameter Update: self._optimizer.step() adjusts weights and biases.
  6. Gradient Reset: Gradients are cleared (zero_grad) before the next batch to prevent accumulation.
- Compilation: The compile method is crucial, taking the loss function and an optimizer (which is initialized by passing it all trainable parameters gathered from the layers).

3. **Key Layer Implementations (layers.py)**
   - Linear (Fully Connected):
     - Implements the core transformation: $y = xW+b$.
     - Uses a build method, where parameter initialization (W, b) is deferred until the input shape is known. This is a common pattern for flexible layer design.
     - Caches the input self._x during the forward pass to be used in the backward pass for calculating ∂W∂L.
     - The gradient update for weights and bias is scaled by the batch size (/ batch_size) to ensure the update represents the average loss gradient across the batch.
   - Flatten:
     - A utility layer that converts multi-dimensional input (e.g., from a future Convolutional layer) into a 2D array, which is required by the Linear layer.
     - The backward pass relies on caching the original shape (self._orig_shape) to correctly reshape the incoming gradient.
4. **Initialization and Optimization**
   - Weight Initialization (utils.py): The library supports Xavier/Glorot and He/Kaiming initialization. This is critical for preventing exploding/vanishing gradients, especially in deep networks.
   - Optimizer (optimizer.py): The SGD optimizer is implemented with support for momentum, which adds a fraction of the previous update vector to the current one, smoothing updates and speeding up convergence.

II. XOR Test Results

Model Architecture for XOR Problem:

Step	Component	Purpose	Input -> Output Shape
Input	X	XOR dataset (4 samples)	(4, 2)
1st Layer	Linear(4)	Learns non-linear feature combinations.	(4, 2) -> (4, 4)
Activation	Tanh()	Introduces non-linearity	(4, 4) -> (4, 4)
2nd Layer	Linear(1)	Maps hidden features to a single output.	(4, 4) -> (4, 1)
Output Activation	Sigmoid()	Final squashing to probability [0, 1].	(4, 1) -> (4, 1)

- **Configuration:**
  - Optimizer: SGD(lr=0.5)
  - Loss: MSELoss
  - Epochs: 350
  - Batch Size: 4
- **Result Analysis:** After 350 epochs, the model successfully converged to the correct mapping.

Input	Expected Output	Model Prediction	Rounded Output
(0,0)	0	0.43	0
(0,1)	1	0.52	1
(1,0)	1	0.52	1
(1,1)	0	0.498	0