FACULTY OF ENGINEERING AIN SHAMS UNIVERSITY

SENIOR-2, MECHATRONICS DEPARTMENT

FALL 25

COMPUTATIONAL INTELLIGENCE

CSE473S

Building a Neural Network Library & Advanced Applications Project's Technical Report

(milestone 1)

Name	Id
Adham Sabry said	2100703
Kholoud Ahmed	2100712
Omar Ayman	2000328
Suhaila Khaled	2001835
Taha Gamal	2100566

Submitted to:

Eng Abdullah

Dr Hossam

1 TABLE OF CONTENTS

2 ABSTRACT

This project presents the design and implementation of a lightweight neural network library developed entirely from scratch using Python and NumPy. The library provides modular components including dense layers, activation functions, loss functions, and an optimization algorithm, enabling users to construct and train simple feedforward neural networks. To validate the correctness and effectiveness of the library, the classic XOR classification problem is used as a benchmarking task. The XOR problem is well known for being non-linearly separable, and therefore requires a multi-layer network to solve—making it an ideal test of forward propagation, backward propagation, gradient computation, and parameter updates. The results demonstrate that the implemented library is capable of learning the XOR function, confirming that the core components function correctly and cohesively.

3 INTRODUCTION

Building a neural network from scratch is a fundamental exercise for understanding the mathematics and internal mechanisms of deep learning. Modern frameworks such as TensorFlow and PyTorch automate almost all aspects of model creation, making it difficult to appreciate how neural networks actually compute outputs, propagate gradients, and update parameters.

This project aims to bridge that gap by implementing a fully functional neural network library without relying on high-level machine learning tools. The library includes modular implementations of dense (fully connected) layers, four activation functions (ReLU, Sigmoid, Tanh, Softmax), two loss functions (Mean Squared Error and Cross-Entropy), and a simple Stochastic Gradient Descent optimizer.

To verify that the system works as intended, the library is evaluated on the classic XOR problem. The XOR task is widely used in literature as a minimal but non-trivial benchmark because it cannot be solved by a single-layer perceptron. Consequently, successfully learning XOR confirms that the library handles linear transformations, non-linear activations, and backpropagation correctly.

4 OBJECTIVES

The main objectives of this project are:

1. Implement a Modular Neural Network Library

- Build core components such as Dense layers, activation functions, loss functions, and an optimizer.

- Ensure each component is designed in a reusable and extensible way, similar to modern deep learning frameworks.

2. Demonstrate Understanding of Forward and Backward Propagation

- Manually compute linear transformations, activations, and gradients.
- Implement derivative formulas required for each activation and loss function.
- Enable end-to-end gradient flow across multiple layers.

3. Validate the Library Using the XOR Problem

- Construct a simple multi-layer network using the developed library.
- Train it on the XOR dataset and verify that the model converges to correct predictions.
- Use the results to confirm that the library performs numerical computation and learning correctly.

4. Provide an Educational Foundation for Future Extensions

- Create a structure that can be expanded later with new layers, optimizers, or training features.
- Lay the groundwork for more complex models such as multi-class classifiers or deeper networks.

5 LIBRARY DESIGN AND ARCHITECTURE CHOICE

5.1 OVERVIEW OF THE LIBRARY ARCHITECTURE

This library is designed as a **modular, extensible neural-network framework**, built from scratch to mimic how real deep-learning libraries (like PyTorch or Keras) are structured. The architecture separates responsibilities into five main components:

- **Activations:** forward/backward output check
- **Dense layer:** forward/backward + gradient check
- **Loss:** forward and backward
- **Optimizer:** weight update check
- **Sequential model:** forward/backward through multiple layers
- **Utils:** check flatten/normalize/one-hot

This separation allows:

- Clean reusable code
- Easy extension (you can add new layers, activations, optimizers)
- A clear training pipeline similar to real-world frameworks

5.1.1 Layer Architecture (Dense Layer)

***Why the Dense layer exists**

A Dense (fully connected) layer transforms inputs as:

$$y = xW + b$$

Design goals:

- Isolate weight storage
- Keep forward and backward passes self-contained
- Make it easy to add more layer types later

***What the Dense class contains**

- **Weights** (W) and **biases** (b)
- **Gradients**
- **Forward pass** (compute output)
- **Backward pass** (compute gradients for training)

5.1.2 Activation Functions

Each activation class implements:

- A **forward** method: computing the activation output from pre-activation inputs,
- A **backward** method: computing gradients (derivatives) wrt its inputs so backprop works properly.

Activation functions should remain **pluggable** — that is, you should be able to choose any of ReLU/Sigmoid/Tanh/Softmax when you define a layer

For hidden layers: **ReLU or Tanh** are usually better than Sigmoid (less gradient vanishing, better training dynamics).

For output layer: choose **Sigmoid** (binary output) or **Softmax** (multi-class) depending on your task.

Activation	Typical Use Case	Advantages / Trade-offs
ReLU	Hidden layers in deep networks, regression or classification	Simple, fast, mitigates vanishing-gradient compared to Sigmoid/Tanh when input > 0. But can "die" (zero gradients) if many inputs negative.
Tanh	Hidden layers when you want zero-centered activations (output in –1 to 1)	Helps symmetry breaking and often converges faster than Sigmoid for hidden layers in small networks.

Activation	Typical Use Case	Advantages / Trade-offs
Sigmoid	Output layer in binary classification / binary-ish tasks, or hidden layers (less common now)	Output between 0 and 1, useful for probabilities. But gradient vanishes when values saturate (close to 0 or 1).
Softmax	Output layer for multi-class classification (more than two classes)	Converts a vector of arbitrary real-valued scores into a probability distribution over classes (all outputs sum to 1).

ReLU (Rectified Linear Unit) → $\mathrm{ReLU}(x) = \max(0, x)$

Sigmoid → $\sigma(x) = 1 / (1 + e^{-x})$ (binary-class)

Tanh → $\tanh(x) = (e^x - e^{-x}) / (e^x + e^{-x})$ (hidden-layers)

SoftMax → $\mathrm{softmax}(z_i) = e^{z_i} / \Sigma_j e^{z_j}$, For an input vector $z = [z_1, z_2, \ldots, z_n]$: (multi)

5.1.3 Loss Function

a. MSE (Mean Squared Error)

Use when:

- Regression tasks
- Educational/simple networks like XOR
- Binary tasks *if using Sigmoid* (still works but not optimal)

Why:

- Easy to compute
- Works with any activation
- Common for teaching backpropagation

Limitation:

- Not ideal for classification
- Slower training when outputs depend on probabilities

b. Cross-Entropy Loss

Use when:

- Multi-class classification (Softmax output)
- Binary classification (Sigmoid output)

Why:

- Works perfectly with Sigmoid and Softmax
- Much faster learning
- Produces stronger gradients
- Standard in all modern neural networks

5.1.4 Optimizer

Why the optimizer is its own class

The optimizer:

- Updates weights using gradients
- Allows future extension (Adam, RMSProp, etc.)

The current optimizer uses **Stochastic Gradient Descent** for clarity.

5.1.5 Sequential Model

Purpose:

The **Sequential** class is the *orchestration engine*.

It:

- Holds layers in order
- Runs forward pass through all layers
- Runs backward pass through all layers
- Calls the optimizer
- Computes loss
- Trains the model across epochs

6 Gradient Checking (Analytical Vs Numerical)

Gradient checking (unit test)

- Pick a very small model or single layer and a small input sample.

- Compute numerical gradient with finite differences for a single weight (or small subset).
- Compare with analytic dW from backward. They should be very close (difference ~1e-7 or so depending on ε).

By using the numerical estimation, which is a standard method, also called gradient checking, finite differences test. It has the formula :

dL/dW ≈ [L(W + ε) - L(W - ε)] / (2ε)

where:

L: The loss function (e.g., MSE, cross-entropy).
This takes inputs → outputs → computes one scalar error.

Example (our code): loss_fn = MSE() → loss=loss_fn.forward(out,y)

W: *One single weight* value inside your network.
Gradient checking works by picking **one element** of the weight matrix (e.g., W[1,2]).

W+ ε : The same weight matrix, except **one weight is slightly increased**.

$$W_{ij}^{+} = W_{ij} + \varepsilon$$

This is used to approximate the slope of the loss when W increases.

W + ε : Same idea, but slightly decreased.

$$W_{ij}^{-} = W_{ij} - \varepsilon$$

ε (epsilon) : A very tiny value, in our code:

$$\varepsilon = 10^{-7}$$

This shifts the weight just a little so we can compute a numerical derivative.

∂W/∂L : This is the true gradient computed by OUR backpropagation code.

NOTE : Numerical gradient is accurate but it is very slow, as it goes one by one across the weights. And if it used for learning, it will take forever. It is only used while debugging.

Analytical gradient: This is the derivative computed by our backpropagation code.

7 THE XOR PROBLEM

Why XOR Is Important in Neural Networks ?

1. A Single-Layer Perceptron Cannot Solve XOR

A single-layer model can only learn functions that are linearly separable.
Since XOR requires a curved or piecewise decision boundary, a network with only one layer cannot represent it.

This limitation was historically important because early neural networks failed on XOR, leading to the "AI winter."

2. A Multi-Layer Network *Can* Solve XOR

When we add **one or more hidden layers** with **non-linear activation functions**, the network gains the capacity to learn non-linear decision boundaries.

A small network such as:

2 inputs → 2 neurons → 1 output

can correctly learn the XOR function.

3. XOR Is the Simplest Non-Linear Benchmark

We use XOR as a test because:

- It is extremely small (easy to visualize and understand).
- It exposes the difference between linear and non-linear models.
- It verifies that the network's **forward pass, backpropagation, activation functions, and weight updates** all work correctly.

- It is a minimal problem that requires **non-linear transformations**—the core strength of neural networks.

Why We Use XOR to Validate a Custom Library?

Implementing a neural-network library from scratch involves many components: weight initialization, matrix multiplication, activations, derivatives, backpropagation, and optimizers.

If the library can successfully learn XOR, it proves that:

- The layers are computing correctly
- Activation functions are applied correctly
- Derivatives are correct
- Backpropagation flows through the network
- The optimizer updates the weights properly
- The model can learn a non-linear function

Because XOR is both challenging and simple, it is the **perfect test case** when building a neural network from scratch.

7.1 TRAINING USING OUR LIBRARY

We are using our custom library `my_library_final`, which includes:

- **Dense** → Fully connected layer with weights, biases, forward and backward methods
- **Sigmoid / Tanh** → Activation functions with forward and backward (derivative) computation
- **MSE** → Mean Squared Error loss
- **SGD** → Optimizer that updates weights and biases
- **Sequential** → Container that orchestrates forward/backward passes through all layers

2 inputs → 4 hidden neurons → 1 output neuron (created network for xor problem)

1. Dataset

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

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

X contains 4 samples, each with 2 features (0 or 1).

y contains the target XOR output for each input.

NOTE : All **4 samples** are processed together in a **single batch**, because there is no batching or shuffle implemented.

2. Layers

```
layers = [
    Dense(2, 4, activation=Sigmoid()),
    Dense(4, 1, activation=Sigmoid())
]
```

- ***First Dense layer***:
  - *Input size = 2*
  - *Output size = 4*
  - *Activation = Sigmoid*
- ***Second Dense layer***:
  - *Input size = 4*
  - *Output size = 1*
  - *Activation = Sigmoid*

NOTE: *Each layer* ***stores weights W, biases b****, and performs* ***forward and backward computations****.*

3. Model Initialization

```
model = Sequential(layers)
```

Sequential takes the list of layers and handles:

- *Forward propagation through all layers*
- *Backward propagation through all layers*

4. Loss function

```
loss_fn = MSE()
```

- *Computes the* ***difference between predicted outputs and targets***
- *Provides* ***gradient of loss with respect to output****, needed for backprop*

5. Optimizer

```
optimizer = SGD(lr=1)
```

- *Stochastic Gradient Descent*
- *Updates each layer's weights and biases using* ***gradients computed during backward pass***
- *Learning rate = 1 (relatively high for this tiny network)*

6. Training Loop

```
for epoch in range(10000):
    out = model.forward(X)          # Forward pass
    loss = loss_fn.forward(out, y)  # Compute loss
    dA = loss_fn.backward()         # Gradient of loss w.r.t output
    model.backward(dA)              # Backprop through all layers
    for layer in layers:
        optimizer.step(layer)       # Update weights and biases
```

7. Sequence of Operations in One Epoch

1. ***Forward pass (model.forward)***
   - *Input X → Layer1 → Sigmoid → Layer2 → Sigmoid → Output*
   - *Produces predicted output for all 4 samples at once*

2. ***Loss computation (loss_fn.forward)***
   - *Compute MSE:*

$$loss = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y_i})^2$$

   - *Here N = 4 samples*

3. ***Backward pass (loss_fn.backward)***
   - *Compute dLoss/dOutput: gradient of loss w.r.t predicted outputs*

4. ***Backpropagation (model.backward)***
   - *Propagate gradient through output layer → hidden layer → input*
   - *Each layer computes:*
     - *dLoss/dW = input^T * dLoss/dOutput*
     - *dLoss/db = sum of gradients over all samples*

5. ***Weight update (optimizer.step)***
   - *Update W and b using gradient and learning rate*

NOTE: *All 4 samples are processed simultaneously (batch size = 4). This is batch gradient descent.*

8. Computations Per Epoch

*For **4 samples, 2-layer network (2→4→1)**:*

1. ***Forward pass***
   - *Layer 1: 4 neurons × 2 inputs × 4 samples → 32 multiplications + 4 biases per sample*
   - *Sigmoid activation: 4 neurons × 4 samples → 16 sigmoid computations*
   - *Layer 2: 1 neuron × 4 inputs × 4 samples → 16 multiplications + 1 bias per sample*
   - *Sigmoid activation: 1 neuron × 4 samples → 4 sigmoid computations*

2. ***Backward pass***
   - *Compute dLoss/dOutput → dLoss/dW and dLoss/db for each layer*
   - *Layer 2: 4 gradients × 1 output → 4 multiplications for dW*
   - *Layer 1: 2 inputs × 4 neurons × 4 samples → 32 multiplications for dW*

3. ***Weight update***
   - *Each weight updated once per epoch using gradient*

Summary: *One epoch = 1 forward pass + 1 backward pass + 1 update step for all weights*

```java
Input X (4 samples, 2 features)
        |
        ▼
Dense Layer 1 (2→4) + Sigmoid
        |
        ▼
Dense Layer 2 (4→1) + Sigmoid
        |
        ▼
Predicted Output (4 samples, 1 feature)
        |
        ▼
Compute Loss (MSE)
        |
        ▼
Compute Gradient dLoss/dOutput
        |
        ▼
Backprop through Layer 2 → Layer 1
        |
        ▼
Optimizer updates W and b
        |
        ▼
Next Epoch
```

a. RESULTS

1. observing the output

2 inputs → 4 hidden neurons → 1 output neuron

Activation: Sigmoid

Loss: MSE

Optimizer: SGD, lr=1

Epochs: 10000

Batch size: 4 (all samples at once)

***Predictions are around**: [[0.011]*

[0.987]

[0.984]

[0.017]]

Target XOR outputs:

[[0],

[1],

[1],

[0]]

2. *analysis of predictions:*

    1. ***Correct classification***

        - *Inputs [0,0] → predicted ≈ 0*
        - *Inputs [0,1] → predicted ≈ 1*
        - *Inputs [1,0] → predicted ≈ 1*
        - *Inputs [1,1] → predicted ≈ 0*

*This confirms the network has successfully **learned the XOR function**.*

    2. ***Confidence of predictions***

        - *Predictions are **very close to 0 or 1**.*
        - *Sigmoid outputs in the range [0.01, 0.99] indicate **strong confidence**.*

    3. ***Loss convergence***

        - *Loss decreases steadily over epochs (as seen in the printed output every 1000 epochs).*
        - *Example: starting loss ≈ 0.25 → final loss ≈ 0.0001–0.0002 (depends on initialization).*
        - *This shows that **gradient computation, backprop, and weight updates** are implemented correctly.*

3. *Why These Results Make Sense*

1. ***Network depth and nonlinearity***

    - *XOR is **non-linear** → cannot be solved by a single-layer perceptron.*
    - *Adding a hidden layer with **4 neurons and sigmoid activation** provides enough capacity for the network to **learn a non-linear boundary**.*

2. ***Learning rate***

    - *lr=1 is high, but for this tiny network and batch of 4 samples, it converges without instability.*

3. ***Batch size***

    - *All 4 samples processed together → **batch gradient descent**.*
    - *This is sufficient for a tiny dataset like XOR; no stochastic noise to slow convergence.*

4. Observations about the Training

- ***Forward pass**: computes outputs through hidden → output layer*
- ***Loss computation**: MSE is very small at the end → model matches targets*
- ***Backward pass**: gradients propagate correctly*
- ***Weight update**: optimizer successfully updates all weights to minimize loss*

*Overall: The network has learned XOR correctly, and the output shows both **correct classification** and **high confidence**.*

5. Optional Improvements / Notes

1. ***Activation functions***

    - *Sigmoid works, but **Tanh** in hidden layer may converge faster (centered at 0).*

2. ***Learning rate***

    - *lr=1 works here, but for larger networks, a smaller lr (0.1 or 0.01) is safer.*

3. ***Hidden neurons***

    - *2 neurons are technically enough for XOR; you used 4, giving the network extra capacity.*

4. ***Loss function***

    - *MSE is okay, but for binary classification, **binary cross-entropy** is usually better and can produce sharper predictions.*

9. TensorFlow For XOR Problem

a. What is TensorFlow

TensorFlow is an open-source machine learning framework created by Google.
It allows you to build, train, and deploy neural networks easily.

You can think of TensorFlow as:

- A **toolbox** for creating neural network layers
- A **math engine** that computes gradients automatically (auto-differentiation)
- A **runtime** that efficiently runs models on CPU/GPU/TPU
- A **high-level API (Keras)** for building neural networks with minimal code

In research projects, TensorFlow (or PyTorch) is often used to **validate custom neural network implementations**, exactly like you're doing by testing your own library on XOR.

b. USED TENSORFLOW FUNCTIONS IN OUR CODE

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation='tanh', input_shape=(2,)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

tf.keras.Sequential: This creates a **linear stack of layers**, meaning each layer feeds into the next.

tf.keras.layers.Dense : This creates a **fully connected (dense) layer**.

First Dense Layer: `tf.keras.layers.Dense(4, activation='tanh',input_shape=(2,))`

- 4 neurons
- activation function = tanh
- input shape = 2 features (the two bits of XOR)

This is the hidden layer.

Second Dense Layer: `tf.keras.layers.Dense(1, activation='sigmoid')`

- 1 output neuron (for XOR output 0 or 1)
- activation = sigmoid (good for binary output)

compiling model:

```
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.075),
    loss='mse'
)
```

Optimizer

SGD = stochastic gradient descent

- adjusts the weights
- learning rate controls how big each update is

loss

mse = mean squared error

- computes how far predictions are from true values
- used to update the weights during training

Training the model: `history = model.fit(X, y, epochs=5000, verbose=0)`

`model.fit()`

This function:

- feeds X through the model
- computes output
- computes loss
- does backpropagation and weight updates
- repeats for 5000 epochs

verbose=0 means "train silently".

history contains loss values you can plot if needed.

Making predictions: `preds = model.predict(X)`

- model.predict() runs a forward pass
- gives the model's learned XOR outputs
- results will be close to 0 or 1 after training

c. RESULTS OF SOLVING XOR PROBLEM WITH TENSORFLOW

```
Predictions:
[[0.02875548]
 [0.936755  ]
 [0.9458372 ]
 [0.06684634]]
```

10. OUR LIBRARY VS TENSORFLOW COMPARISON

a. PREDICTIONS COMPARISON

Input	Target	Custom Library Prediction	TensorFlow Prediction
[0,0]	0	0.0110	0.0288
[0,1]	1	0.9870	0.9368
[1,0]	1	0.9837	0.9458
[1,1]	0	0.0167	0.0668

Observations:

1. **Accuracy**

- Both models correctly predict XOR outputs:
    - Inputs [0,0] and [1,1] → near 0
    - Inputs [0,1] and [1,0] → near 1
- Both are clearly **solving XOR**.

2. **Closer to Target**
    - Your custom library predictions are **closer to the true 0/1 values**:
        - Example: [0,1] → 0.987 vs TensorFlow's 0.937
        - [1,1] → 0.017 vs TensorFlow's 0.067
    - This shows your library learned slightly **more confident outputs** for this tiny dataset.
3. **TensorFlow Predictions**
    - TensorFlow outputs are slightly "smoother" / less extreme.
    - Likely due to:
        - Different weight initialization
        - Learning rate
        - Random seed
        - Fewer epochs or optimizer dynamics
4. **Overall**
    - Both models are functionally correct.
    - Differences are minor and expected given:
        - Small dataset (4 samples)
        - Random initialization
        - Non-deterministic SGD updates

b. COMMENTARY TO RESULTS

- **Validation:** The comparison confirms that your **custom library is working correctly**, as it produces nearly perfect XOR outputs.
- **Confidence:** The closer predictions to 0 and 1 indicate that your custom network learned strong decision boundaries.
- **TensorFlow Differences:** TensorFlow is slightly more conservative due to default initialization and internal optimizations; for larger datasets, these differences average out.
- **Conclusion:** Both approaches validate your library's forward propagation, backward propagation, and optimizer implementation. The small differences are **normal and expected** in neural networks trained on tiny datasets.

11. CHALLENGES AND LESSONS

11.1 CHALLENGES FACED

1. Implementing Backpropagation Correctly

Backpropagation is mathematically dense: it requires understanding partial derivatives, chain rule, matrix operations, and how errors propagate backward through layers.

Small mistakes (shape mismatches, wrong transpose, wrong derivative) can break training completely.

2. Handling Weight Initialization

The model's ability to learn depended heavily on how weights were initialized.

Too small → learning becomes slow.

Too large → activations saturate (sigmoid/tanh stop updating).

3. Gradient Vanishing with Sigmoid

Using sigmoid activation in multiple layers caused gradients to shrink, especially early in training.

This made convergence slower and required careful tuning of learning rate.

4. Choosing the Right Learning Rate

A learning rate that is:

- too high → training diverges
- too low → loss improves extremely slowly

Finding a stable value that works for XOR required experimentation.

5. Ensuring Consistent Matrix Shapes

Forward and backward passes require precise dimension alignment.

Mismatched shapes in dot products, biases, or gradients caused runtime errors and required careful debugging.

6. Verifying Correctness Without a Framework

Since you were not using TensorFlow/PyTorch, you had to create your own tools to:

- print intermediate outputs
- inspect weights
- compare with expected derivatives

Debugging this without built-in tools was challenging but educational.

7. Designing a Modular, Extensible Library

Creating a clean structure (Layers, Activations, Losses, Optimizers, Sequential model) required architectural planning.

Incorrect or messy design made it harder to reuse components.

12. REFERENCES

github repo: jaegerattacks-lgtm/ci_project