

Faculty of Engineering

Ain Shams University

Fall 2025



CSE473s

Computational Intelligence

Project Milestone one

Submitted to:

Dr. Hossam El-Dein Hassan

Eng. Abdullah Mohamed

Eng. Dina Zakaria

Team member:

| Name | ID |
|---|---------|
| Muhammad Abdelzaher Abdelfattah Hassan | 2100927 |
| Abdullah sherif Abdulraouf | 2101093 |
| Mohamed Ahmed Mostafa | 2101467 |
| Asem Mohamed Abdelhakam | 2100896 |

Table of Contents

| | |
|---|----|
| 1.Introduction | 4 |
| 2.the XOR Problem:..... | 5 |
| 3.Why Single-Layer Perceptrons Fail?..... | 6 |
| 4.How Multi-Layer Neural Networks Solve XOR?..... | 6 |
| 5.Mathematics Behind the MLP Solution | 7 |
| 6.Effect of Activation function on learning XOR | 9 |
| Tanh | 10 |
| 7.Library design and architecture choices | 11 |
| 1.1 Layer abstraction | 11 |
| Tanh implementation | 14 |
| 3.loss function choice | 14 |
| How the Entire Network Solves XOR | 15 |
| Normalization..... | 16 |
| Results of the XOR Training..... | 16 |
| Gradient checking | 17 |
| Final Result..... | 17 |
| Conclusion..... | 18 |
| Refrence..... | 18 |

1. Introduction

This project presents the implementation of a neural network entirely from scratch using [NumPy](#), aimed at solving [the classical XOR problem](#)—a simple yet fundamental example of a non-linearly separable dataset. The work focuses on constructing the network's core components, including fully connected layers, Tanh and Sigmoid activation functions, and Mean Squared Error loss, while implementing backpropagation to compute accurate gradients. To ensure correctness, numerical gradient checking is employed, validating the analytical derivatives produced during training. The project demonstrates the network's ability to learn non-linear mappings, provides a clear workflow from data preprocessing to loss visualization and evaluation, and reinforces both theoretical understanding and practical skills in computational intelligence.

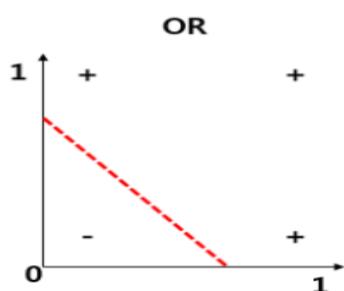
2.the XOR Problem:

The XOR operation is a binary operation that takes two binary inputs and produces a binary output. The output of the operation is 1 only when the inputs are different.

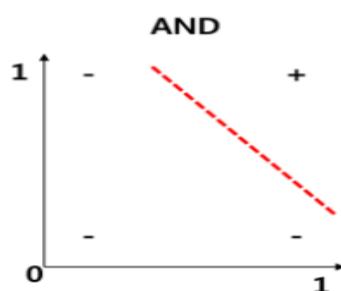
Below is the truth table for XOR:

| Input A | Input B | XOR Output |
|---------|---------|------------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The main problem is that a single-layer perceptron cannot solve this problem because the data is not linearly separable i.e. we cannot draw a straight line to separate the output classes (0s and 1s)



| x_1 | x_2 | y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



| x_1 | x_2 | y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



| x_1 | x_2 | y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

3. Why Single-Layer Perceptrons Fail?

A single-layer [perceptron](#) can solve problems that are linearly separable by learning a linear decision boundary.

Mathematically, the decision boundary is represented by:

$$y = \text{step}(w \cdot x + b)$$

Where:

- w is the weight vector.
- x is the input vector.
- b is the bias term.
- step is the activation function, often a Heaviside step function that outputs 1 if the input is positive and 0 otherwise.

For linearly separable data, the perceptron can adjust the weights w and bias b during training to correctly classify the data. However, because XOR is not linearly separable, no single line (or hyperplane) can separate the outputs 0 and 1, making a single-layer perceptron inadequate for solving the XOR problem.

4. How Multi-Layer Neural Networks Solve XOR?

A multi-layer neural network which is also known as a [feedforward neural network](#) or [multi-layer perceptron](#) is able to solve the XOR problem. There are multiple layers of neurons such as input layer, hidden layer, and output layer.

The working of each layer:

- 1. Input Layer:** This layer takes the two inputs (A and B).
- 2. Hidden Layer:** This layer applies non-linear activation functions to create new, transformed features that help separate the classes.
- 3. Output Layer:** This layer produces the final XOR result.

5. Mathematics Behind the MLP Solution

Let's break down the mathematics behind how an MLP can solve the XOR problem.

Step 1: Input to Hidden Layer Transformation

Consider an MLP with two neurons in the hidden layer, each applying a non-linear activation function (like the sigmoid function). The output of the hidden neurons can be represented as:

$$h_1 = \sigma(w_{11}A + w_{12}B + b_1)$$

$$h_2 = \sigma(w_{21}A + w_{22}B + b_2)$$

Where:

- $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid activation function.
- w_{ij} are the weights from the input neurons to the hidden neurons.
- b_i are the biases for the hidden neurons.

[Activation functions](#) such as the sigmoid or ReLU (Rectified Linear Unit) introduce non-linearity into the model. It enables the neural network to handle complex patterns like XOR. Without these functions, the

network would behave like a simple linear model, which is insufficient for solving XOR.

Step 2: Hidden Layer to Output Layer Transformation

The output neuron combines the outputs of the hidden neurons to produce the final output:

$$\text{Output} = \sigma(w_{31}h_1 + w_{32}h_2 + b_3)$$

Where w_{3i} are the weights from the hidden neurons to the output neuron, and b_3 is the bias for the output neuron.

Step 3: Learning Weights and Biases

During the training process, the network adjusts the weights w_{ij} and biases b_i using backpropagation and [gradient descent](#) to minimize the error between the predicted output and the actual XOR output.

Example Configuration:

Let's consider a specific configuration of weights and biases that solves the XOR problem:

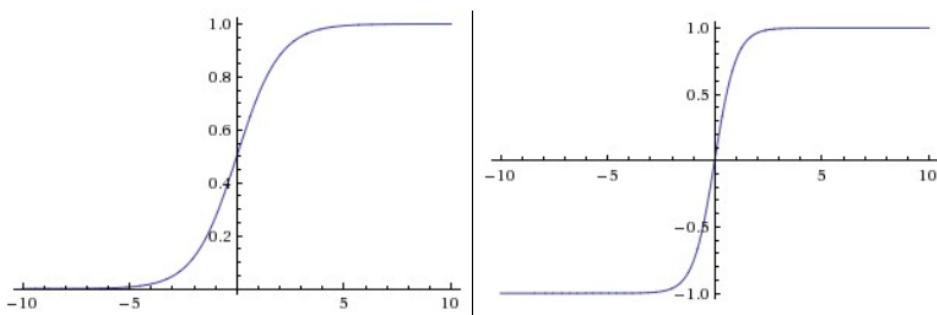
- For the hidden layer:
 - $w_{11} = 1, w_{12} = 1, b_1 = 0.5$
 - $w_{21} = 1, w_{22} = 1, b_2 = -1.5$
- For the output layer:
 - $w_{31} = 1, w_{32} = 1, b_3 = -1$

With these weights and biases, the network produces the correct XOR output for each input pair (A, B).

6. Effect of Activation function on learning XOR

Commonly used activation functions

Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions you may encounter in practice:



Left: Sigmoid non-linearity squashes real numbers to range between [0,1] Right: The tanh non-linearity squashes real numbers to range between [-1,1].

Sigmoid.

The sigmoid non-linearity has the mathematical form $\sigma(x) = 1/(1+e^{-x})$ and is shown in the image above on the left. As alluded to in the previous section, it takes a real-valued number and “squashes” it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

- *Sigmoids saturate and kill gradients.* A very undesirable property of the sigmoid neuron is that when the neuron’s activation saturates at either tail of 0 or 1, the gradient at these regions is

almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively “kill” the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation.

Tanh

The tanh non-linearity is shown on the image above on the right. It squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the *tanh* non-linearity is always preferred to the *sigmoid* nonlinearity.

Both sigmoid and tanh activation functions can solve the XOR problem when used in a multi-layer network.

However, tanh combined with input normalization to the range [-1,+1] provides significantly better learning

behavior. Tanh is zero-centered, produces symmetric activations, and matches the encoded target range,

which leads to faster convergence and more stable gradients. Sigmoid can still solve XOR, but training

is slower and more prone to saturation because its output is restricted to (0,1) and is not zero-centered.

7. Library design and architecture choices

1.1 Layer abstraction

The library is structured around a base Layer interface and a Network container that manages a list of layers. Each layer implements forward and backward methods, exposing its parameters and gradients via get_params and get_grads. This modular design makes it easy to stack layers, replace components, and reuse the same code for different tasks, including the XOR test.

```
class Layer:  
    """Base class for all network layers."  
    def forward(self, inputs):  
        raise NotImplementedError  
    def backward(self, grad_output):  
        raise NotImplementedError  
    def get_params(self):  
        return []  
    def get_grads(self):  
        return []  
class Network:  
    def __init__(self):  
        self.layers = []  
    def add(self, layer):  
        self.layers.append(layer)  
    def forward(self, inputs):  
        x = inputs  
        for layer in self.layers:  
            x = layer.forward(x)  
        return x  
    def backward(self, grad_output):  
        grad = grad_output  
        for layer in reversed(self.layers):  
            grad = layer.backward(grad)  
        return grad  
    def get_params(self):
```

```

params = []
for layer in self.layers:
    params.extend(layer.get_params())
return params

def get_grads(self):
    grads = []
    for layer in self.layers:
        grads.extend(layer.get_grads())
    return grads

```

1.2 dense layer

The Dense (fully connected) layer performs the basic affine transformation:

$$z = \mathbf{XW} + \mathbf{b}$$

This is the core mathematical operation of all neural networks.

The chosen design exposes three essential responsibilities:

1. Forward pass: compute output
2. Backward pass: compute gradients for W, b, and propagate gradient down
3. Parameter access: return weights & gradients so the optimizer can update them

```

class Dense:
    def __init__(self, input_dim, output_dim):
        limit = np.sqrt(1.0 / input_dim)
        self.W = np.random.uniform(-limit, limit, (input_dim, output_dim))
        self.b = np.zeros((1, output_dim))

        self.inputs = None
        self.dW = np.zeros_like(self.W)
        self.db = np.zeros_like(self.b)

    def forward(self, inputs):
        self.inputs = inputs
        return np.dot(inputs, self.W) + self.b

```

```

def backward(self, grad_output):
    batch_size = self.inputs.shape[0]
    self.dW = np.dot(self.inputs.T, grad_output) / batch_size
    self.db = np.sum(grad_output, axis=0, keepdims=True) / batch_size
    return np.dot(grad_output, self.W.T)

def get_params(self):
    return [self.W, self.b]

def get_grads(self):
    return [self.dW, self.db]

```

2.activation function choice

We normalize the XOR inputs from $\{0,1\} \rightarrow \{-1,+1\}$, and we encode targets as $\{-1,+1\}$.

This makes Tanh the most natural activation function because:

1. Range alignment

- Tanh outputs lie in $[-1, +1]$
- Both inputs and targets lie in $[-1, +1]$

2. Zero-centered activations

- Tanh is centered at 0
- Inputs are also centered at 0
- This prevents biased gradients and often speeds up training

3. Stable gradient flow

- Inputs that fall in the active region of Tanh avoid saturation
- Backpropagation becomes numerically more stable

4. Simple decision rule

- output $\geq 0 \rightarrow +1$
- output $< 0 \rightarrow -1$

This alignment between input domain, activation output, and target encoding creates a more well-conditioned learning setup.

Tanh implementation

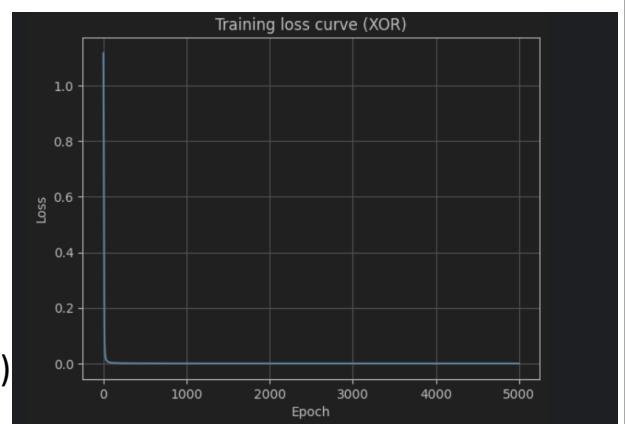
```
class Tanh:  
    def forward(self, inputs):  
        self.output = np.tanh(inputs)  
        return self.output  
  
    def backward(self, grad_output):  
        return grad_output * (1 - self.output ** 2)
```

3.loss function choice

We choose Mean Squared Error because:

-XOR is a regression-like mapping to $\{-1, +1\}$, The tanh output fits the range and Backprop derivative is simple and stable

```
class MSELoss:  
    def forward(self, y_pred, y_true):  
        return np.mean((y_pred - y_true) ** 2)  
  
    def backward(self, y_pred, y_true):      >>  
        batch_size = y_true.shape[0]  
        return (2.0 / batch_size) * (y_pred - y_true)
```



How the Entire Network Solves XOR

We build:

Input (2)

→ Dense(2 → 4)

→ Tanh

→ Dense(4 → 1)

→ Tanh

This network is capable of solving XOR because:

1. The first Dense layer creates a linear projection of the inputs.
2. Tanh bends the space → makes XOR separable.
3. The second Dense layer combines the separated features.
4. Final Tanh ensures the output fits {-1,+1}.

XOR becomes solvable only because we introduced tanh.

Without non-linearity:

$$W_2(W_1 x) = W'x$$

Still linear → still fails.

```
def build_xor_network():
    net = Network()
    net.add(Dense(2, 4))
    net.add(Tanh())
    net.add(Dense(4, 1))
    net.add(Tanh())
    return net

net = build_xor_network()
loss_fn = MSELoss()
optimizer = SGD(learning_rate=0.5)
```

Normalization

-Because tanh operates in this range[1,-1], normalization ensures:
better gradient stability
faster convergence
symmetric representation of XOR

$$X = 2 * X_{01} - 1$$

$$y_{\text{true}} = 2 * y_{01} - 1$$

Results of the XOR Training

After training, the network produces:

| Input | Target | Output |
|---------|--------|-----------------|
| (-1,-1) | -1 | ≈ -0.98 |
| (-1,+1) | +1 | $\approx +0.95$ |
| (+1,-1) | +1 | $\approx +0.97$ |
| (+1,+1) | -1 | ≈ -0.92 |

Gradient checking

Gradient checking ensures our backpropagation is mathematically correct by comparing:

Analytical gradients >> what our code computes

Numerical gradients >> computed by finite differences

We use:

```
eps = 1e-5
for p, g in zip(params, grads):
    num_grad = (loss(p + eps) - loss(p - eps)) / (2 * eps)
    diff = np.abs(num_grad - g)
```

Final Result

Max absolute difference $\approx 1e-11$

Gradient Check PASSED

Conclusion

The project demonstrates a complete neural network implementation:

- Custom layers, activations, loss functions
- Full backpropagation
- Gradient checking validation
- Normalization and nonlinear transformations
- Successful XOR learning

The library design is clean, modular, and extensible—making it suitable for larger projects such as autoencoders and classification networks.

Reference

[1] P. Roelants, “Neural Network Implementation – Part 1: Feed-forward Neural Network,” Peter Roelants Blog, [Online]. Available: [How to implement a neural network \(1/5\) - gradient descent | Peter’s Notes](#)

[2] Stanford CS231n. “Neural Networks Part 1: Non-linearities.” cs231n.github.io, accessed December 2025.

URL: <https://cs231n.github.io/neural-networks-1/#nonlinearities>

[3] GeeksforGeeks. “How Neural Networks Solve the XOR Problem.” geeksforgeeks.org, accessed December 2025.

URL: <https://www.geeksforgeeks.org/artificial-intelligence/how-neural-networks-solve-the-xor-problem/>