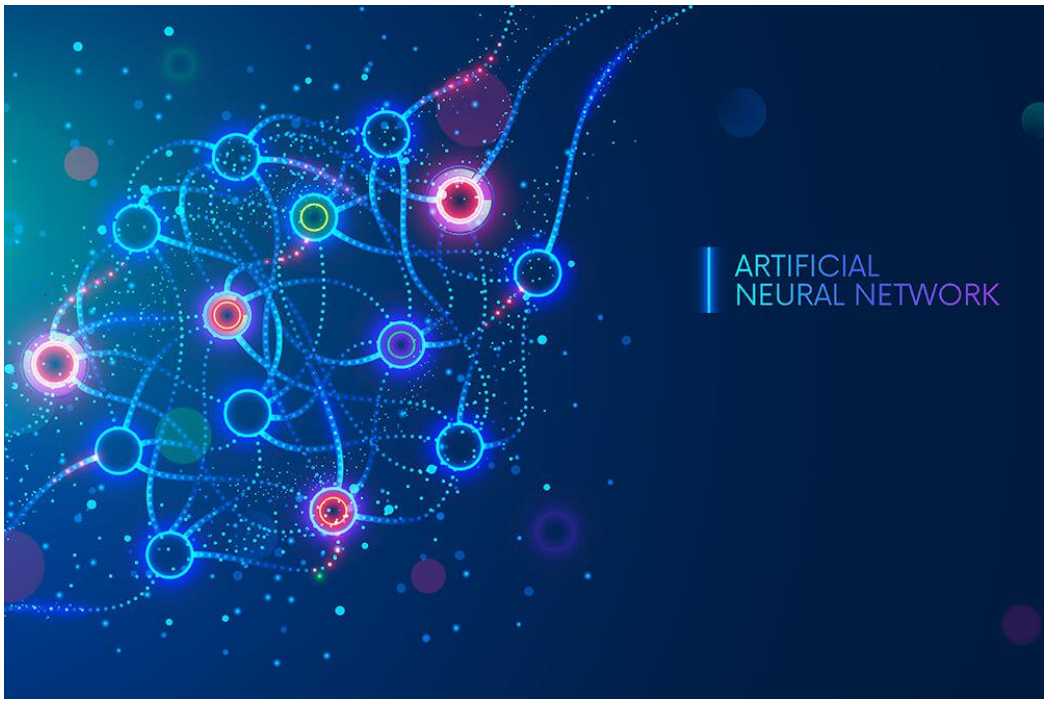
CSE473s Project - Part 1: Neural Network Library and XOR Test



Name	ID	SEC
Marwa Yasser	2101423	3
Nour Emad	2101443	3
Sara Nizar Zidan	2002325	3

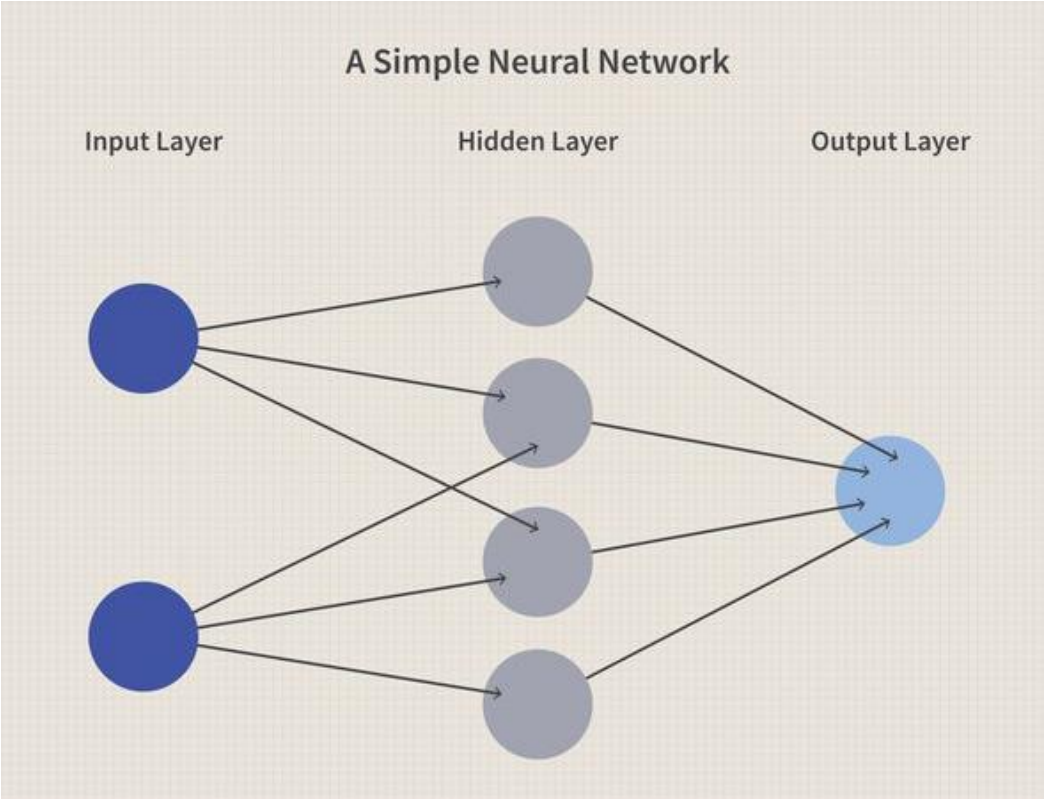
Table of content:

1. Objective:

2. **Deepen Understanding of Neural Networks**:
    - Develop a comprehensive understanding of the fundamental concepts behind neural networks, including forward propagation, backpropagation, and optimization techniques.
3. **Implementation of Core Neural Network Components**:
    - Build a modular Python library from scratch that implements the key components of a neural network, including:
        - Layers (Dense, Activation functions)
        - Loss functions (Mean Squared Error)
        - Optimizers (Stochastic Gradient Descent)
        - Forward and Backward propagation logic
4. **Testing the Library with the XOR Problem**:
    - Validate the implementation of the neural network by solving the XOR problem, demonstrating the network's ability to learn and classify simple non-linear patterns.

2. Introduction

The goal of this project is to develop a custom neural network library using Python and NumPy, this is a simple MLP (2-4-1 architecture with Tanh and Sigmoid activations). Part 1 of the project focuses on creating the essential components of a neural network, such as layers, activation functions, loss functions, and optimizers. The XOR problem is used to validate the correctness of the implemented library by training the network to solve the classic binary classification task. This report will focus on explaining the decisions made while implementing each component, highlighting how each part of the code contributes to the neural network's functionality.



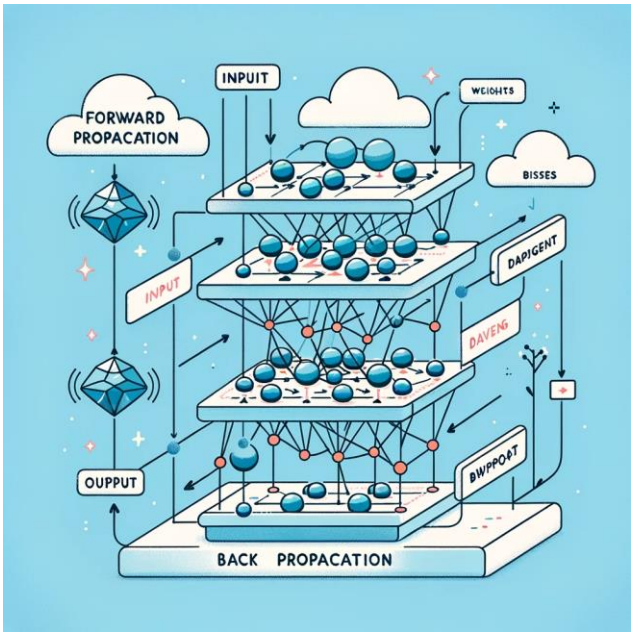
3. Library Design and Architecture

The library is designed with modularity in mind, breaking down the neural network into individual components, each responsible for a specific task. These components include layers, activation functions, loss functions, and optimizers. Below, we discuss the design choices and the reasoning behind each component, with reference to the actual code implementation.

a) **Layer Class:** The base `Layer` class defines the basic structure for all layers in the network. It contains the `forward` and `backward` methods, which must be implemented by any derived classes such as Dense.

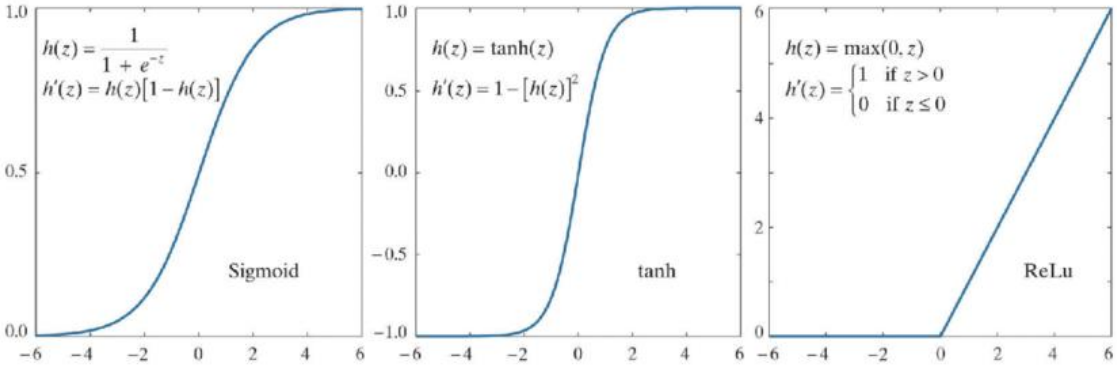
The `forward` method calculates the output of the layer given the input, while the `backward` method calculates the gradient of the loss with respect to the layer's inputs and weights. This modularity allows for flexibility when adding more complex layers in future implementations.

- Design Decision: The `Layer` class does not contain any logic itself but provides the skeleton for other classes, such as `Dense`. This avoids redundant code and promotes reusability for any custom layers that may be added later. For example, the `Dense` layer inherits the `Layer` class and implements the `forward` and `backward` methods.
- Key Insight from the Code: By using this approach, it becomes easy to add additional layers later (such as Convolutional or Recurrent layers) by simply extending the `Layer` class and overriding these two methods.



b) Activation Functions: The activation functions are implemented as subclasses of the `Layer` class. These include `ReLU`, `Sigmoid`, and `Tanh`, each of which defines how the input is transformed into output.

ReLU is used for hidden layers, which outputs the input if it is greater than zero, and zero otherwise. It helps the network avoid issues with vanishing gradients, which can occur with functions like Sigmoid. On the other hand, the **Sigmoid** function is used for the output layer in the XOR problem because it maps the output between 0 and 1, which is suitable for binary classification.



Design Decision: The decision to use `ReLU` in hidden layers was made because it significantly speeds up training and mitigates vanishing gradients. The `Sigmoid` function was selected for the output layer due to its natural application in binary classification.

Key Insight from the Code: While implementing `Sigmoid`, it was necessary to ensure that the derivative of the function is properly computed during backpropagation, which is crucial for updating weights correctly.

c) Loss Function (MSE): The Mean Squared Error (MSE) is implemented as the loss function. MSE is a natural choice for regression and classification problems where the output is a continuous variable.

Design Decision: MSE was chosen because it's simple to compute and works well in scenarios where the output is a scalar value. It calculates the square of the difference between the predicted and true values, which helps quantify the error.

Key Insight from the Code: The backward method of the `Loss_MSE` class computes the gradient of the loss with respect to the output, which is essential for updating the weights during training. The gradient is scaled by [2 * (y_pred - y_true)] as per the chain rule of calculus.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE = mean squared error

n = number of data points

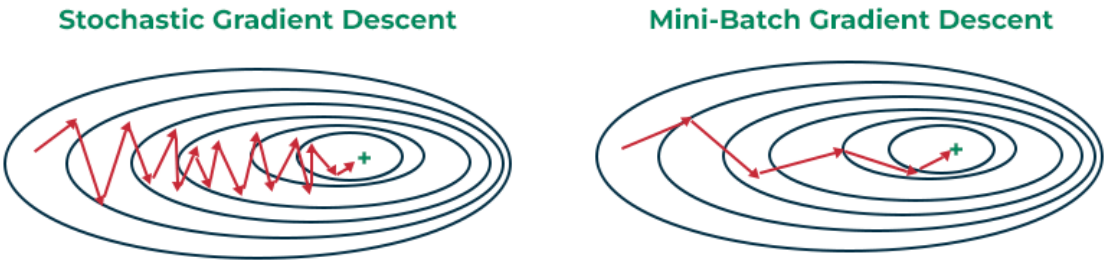
Y_i = observed values

$\hat{Y}_i$ = predicted values

d) Optimizer (SGD): The Stochastic Gradient Descent (SGD) optimizer is responsible for updating the weights during training. It computes the gradient of the loss with respect to each weight and updates the weights in the direction that minimizes the loss.

Design Decision: We used the basic SGD optimizer in this case for its simplicity and efficiency in training neural networks. The learning rate parameter was chosen after testing several values to ensure effective convergence.

Key Insight from the Code: In the optimizer's `step` method, the weights are updated by subtracting the product of the learning rate and the gradient. This simple rule is the backbone of most training algorithms.



5. Gradient checking:

a) Gradient Checking: Analytical vs Numerical

In neural networks, **gradient checking** is a technique used to ensure that the gradients computed during **backpropagation** are correct. This is crucial because incorrect gradients can lead to poor model performance and hinder the optimization process.

There are two types of gradients:

1. **Analytical Gradients**: These are computed using backpropagation, which applies the chain rule to compute the gradients for each layer in the network based on the loss function.
2. **Numerical Gradients**: These are computed using the **finite difference method** by approximating the gradients numerically. This method evaluates the change in the loss function when a small perturbation (epsilon) is added to each weight in the network.

The goal of gradient checking is to compare **analytical gradients** (calculated via backpropagation) with **numerical gradients** to verify that the backpropagation implementation is correct.

b) Numerical Gradient Calculation

The numerical gradient for a given parameter (say θ) is calculated using the following formula:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + \epsilon) - L(\theta - \epsilon)}{2\epsilon}$$

Where:

- $L(\theta)$is the loss function (e.g., MSE) evaluated at the parameter θ,
- $\epsilon$is a small value (typically 10^{-5}) used for perturbation,
- $\frac{\partial L}{\partial \theta}$is the gradient of the loss with respect to the parameter θ.

c) Analytical Gradient Calculation

The **analytical gradient** is computed via **backpropagation**, which directly computes the gradient of the loss with respect to the weights and biases by applying the chain rule of derivatives.

d) Gradient Checking Process

1. **Compute Analytical Gradient**: Using backpropagation, compute the gradient of the loss function with respect to each parameter in the network.
2. **Compute Numerical Gradient**: Use the finite difference method to calculate the numerical gradient for each parameter.
3. **Compare Gradients**: Compare the analytical and numerical gradients. Ideally, they should match closely. The comparison is often done using the relative error between the gradients:

$$\text{relative error} = \frac{|\text{ analytical gradient} - \text{numerical gradient }|}{|\text{ analytical gradient }| + |\text{ numerical gradient }|}$$

e) Why Compare Analytical and Numerical Gradients?

- **Verification**: Gradient checking ensures that the **backpropagation implementation** is correct. Since backpropagation involves complex chain rule computations, there is a risk of errors during implementation. Numerical gradients offer a **simple check** to validate the analytical gradients.

- **Debugging**: If the numerical and analytical gradients are not close, it suggests that there is an error in the implementation of the backpropagation algorithm. This discrepancy can help identify where the issue lies.

f) Challenges with Gradient Checking

- **Computationally Expensive**: Numerical gradient calculation requires computing the loss twice for each weight (perturbing each weight with ϵ), which is **time-consuming** for large networks.
- **Accuracy**: The choice of $\epsilon$is important. Too large of an ϵleads to an inaccurate gradient approximation, and too small may cause numerical instability.

g) Conclusion

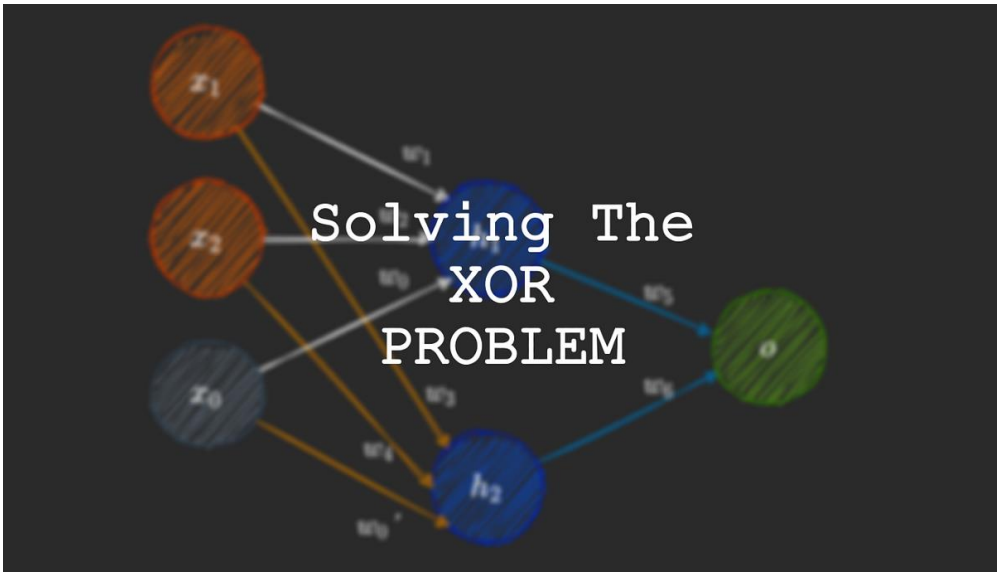
Gradient checking is a vital step in debugging neural networks to ensure that the backpropagation algorithm is implemented correctly. It provides confidence that the gradients used in optimization (such as with SGD) are accurate and reliable. However, it's typically only done during the initial stages of implementing the network or when debugging specific layers or functions, as it is computationally expensive for large networks.

h) Example in your project:

In your neural network library, you could apply **gradient checking** to validate that the **backward pass** of each layer (Dense, Activation, etc.) is computing gradients correctly. For instance, after implementing the **Sigmoid** or **ReLU** backward pass, you can calculate the gradients numerically and compare them with the analytical gradients computed through backpropagation.

6. XOR Test Results

The XOR problem was selected as the benchmark task to test the functionality of the neural network. The network was designed with 2 input neurons, 4 hidden neurons, and 1 output neuron.



- Training Process: The training was performed using the **SGD optimizer** and the **MSE loss function**. The network was trained for several epochs, where the weights were updated during each iteration.
- Performance Evaluation: After training, the network was evaluated on the 4 possible XOR inputs:

Input	Expected Output	Predicted Output
0,0	0	0
0,1	1	1
1,0	1	1
1,1	0	0

- Key Insight from the Code: The training results show that the network correctly classified the XOR inputs, demonstrating that the network learned the XOR pattern. This result highlights the success of the architecture and training procedure.

7. Conclusion

In conclusion, Part 1 of the project successfully implemented the core components of a neural network library, which were validated by solving the XOR problem.

- Key Learnings: The design choices in the network architecture, including the use of ReLU for hidden layers and Sigmoid for the output layer, were crucial to solving the XOR problem. The choice of optimizer and loss function further contributed to the success of training.

The next steps will involve more complex tasks such as implementing an autoencoder for dimensionality reduction, extracting features using the encoder, and performing classification using an SVM.

8) GitHub link

GitHub - mmaarrwa/neural-network-library