# Neural Network Design for Function Approximation

## Objective

The goal of this project was to implement a neural network from scratch using NumPy to approximate a nonlinear function: the Rosenbrock function, which is known for its challenging landscape with a narrow, curved valley. This function is commonly used to test optimization algorithms. The neural network's task was to learn this function based on training data and generalize well to test data.

## Rosenbrock Function

The Rosenbrock function is defined as: f (x1, x2) =(1−x1) ^2+100·(x2−x1^2) ^2 This function has a minimum at x= (1,1) x = (1, 1) x= (1,1), making it a useful test function for neural networks and other optimization techniques.

## Data Generation

- **Training Set:** Generated 200 random points in the range [-1, 1] for both x1 and x2, and calculated the function values.
- **Test Set:** Generated 100 random points in the range [-1, 1] for both x1 and x2 , and calculated their corresponding function values.

The network is trained on the training set, and predictions are tested on the test set.

## Neural Network Architecture

The network is a simple feedforward neural network with the following structure:

- **Input Layer:** 2 neurons (corresponding to x1 and x2 inputs).
- **Hidden Layer:** 70 neurons, which were experimentally chosen to balance the approximation capacity and computational efficiency.
- **Output Layer:** 1 neuron, representing the predicted function value.

## Activation Functions

- **Hidden Layer Activation (ReLU):** Chose the ReLU (Rectified Linear Unit) activation for the hidden layer. ReLU helps introduce non-linearity, allowing the network to approximate the complex non-linear behavior of the Rosenbrock function. ReLU also mitigates the vanishing gradient problem often encountered with sigmoid or tanh activation functions.
- **Output Layer Activation (ReLU):** Used ReLU as the output layer activation. Initially, absolute activation was considered for the output layer, but ReLU was kept for simplicity and to ensure consistency with the hidden layer. This choice is not typical for regression tasks, but for this specific function range and approximation, it allowed reasonable performance.

## Initialization of Weights and Biases

- **Weights:** Initialized weights using Xavier initialization, which scales the weights according to the input size. I have used **XAVIER Initialization** This initialization was chosen to help the network start with balanced gradients across layers.
- **Biases:** Initialized biases randomly from a normal distribution, scaled by 0.01 to keep initial bias values small and avoid overly large activations at the beginning of training.

## Learning Rate and Epochs

- **Learning Rate:** Set to a relatively small value of 0.0008. The learning rate was chosen experimentally to balance convergence speed and stability, as the network was sensitive to larger values, leading to gradient explosion.
- **Epochs:** Set to 25,001. The number of epochs was chosen to allow sufficient training without excessive computation, balancing time efficiency with the need to reach convergence.

## Forward and Backward Propagation

## Forward Propagation

The forward pass computes activations through the network layers as follows:

1. **Hidden Layer:** z= ReLU (X· W_input_to_hidden+ b_hidden)
2. **Output Layer:** yhat = ReLU (z·W_hidden_to_output + b_output)

## Loss Function

The loss function used is **Mean Squared Error (MSE)**, This loss function is common for regression tasks, as it penalizes larger errors more severely, which can help reduce the influence of outliers in predictions.

## Backpropagation

The backpropagation algorithm calculates gradients for updating weights and biases:

1. **Error Calculation:** e= yhat − Y
2. **Output Layer Gradients:** Calculated using the derivative of the ReLU function for the output layer.
3. **Hidden Layer Gradients:** Calculated using the derivative of the ReLU function for the hidden layer, with respect to the error propagated backward.

## Parameter Update with Gradient Clipping

To prevent gradient explosion, a **gradient clipping** mechanism was included, which limits the gradients to a maximum magnitude. Gradient clipping is especially useful when using ReLU activations, which can lead to high variance in gradients if not managed properly.

**Results and Evaluation**

After training, the network was evaluated using the test set with the following metrics:

- **Mean Squared Error (MSE):** Measures the average of the squared differences between predicted and actual values, indicating how well the model fits the data.
- **Mean Absolute Error (MAE):** Measures the average absolute differences between predicted and actual values, giving an idea of the average deviation.

## Observations

1. **Loss at Each Epoch:** The loss generally decreased with each epoch, indicating that the network was learning the function and adjusting parameters effectively.
2. **MSE and MAE on Test Set:** These values provided an indication of the network's generalization ability. A low error would indicate effective learning and generalization, though challenges in function approximation, like the Rosenbrock function's non-linearity, may prevent perfect accuracy. We got an MSE of around 3 and MAE of around 1.5.

## Conclusion

This project demonstrates how a simple neural network can approximate complex non-linear functions when carefully configured. The choices of architecture, activation functions, learning rate, and regularization techniques (gradient clipping) allowed for a stable training process and reasonable function approximation capability.

Further improvements could be made with more complex architectures or advanced techniques like batch normalization, adaptive learning rates, or additional hidden layers to capture more intricate patterns within the function.