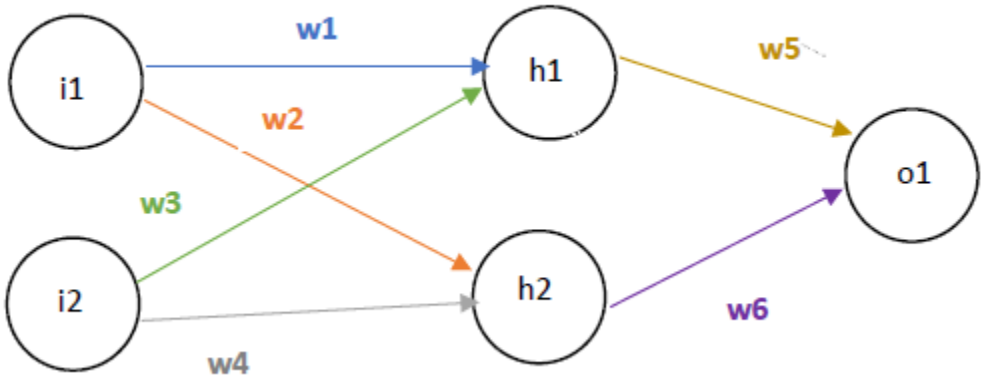
Homework 3 Report: Feedforward Neural Network for XOR Logic Function

OVERVIEW

This homework involves implementing a feedforward neural network (FFNN) from scratch in Python using only NumPy and Matplotlib to solve the XOR logic function. XOR is a classic example of a problem that is not linearly separable, making it a useful exercise for understanding the capabilities of neural networks, especially with hidden layers.

The FFNN designed for this task consists of two input neurons, two hidden neurons, and one output neuron, with sigmoid activation functions. The network was trained using both batch and stochastic gradient descent (SGD), and its performance with and without bias nodes was evaluated.

For the sake of simplicity, the following neural network architecture was used:



METHODOLOGY

- **Dataset:** XOR truth table with two binary inputs and four input-output pairs.
- **Network Architecture:** 2 input nodes → 2 hidden nodes → 1 output node (2-2-1 architecture).
- **Activation Function:** Sigmoid for hidden and output layers.
- **Loss Metric:** Mean Absolute Error (MAE) for simplicity and interpretability.
- **Training Methods:** Both batch and stochastic gradient descent were implemented.
- **Bias Nodes:** Tested the neural network's performance with and without bias nodes.

IMPLEMENTATION

- **Programming Tools**: Python with NumPy for numerical operations and Matplotlib for plotting.
- **Initialization**: Random weights and biases initialized using NumPy with a fixed seed for reproducibility.
- **Training**: Implemented both Batch Gradient Descent (BGD) and Stochastic Gradient Descent (SGD). BGD accumulates gradients over all samples before updating weights, SGD updates weights after each sample.
- **Feedforward**: Calculates hidden layer outputs and final outputs using the sigmoid activation function.
- **Backpropagation**: Computes gradients of the loss with respect to weights and biases, using the chain rule and sigmoid derivative.
- **Weight Updates**: Applied using calculated gradients scaled by the learning rate.
- **Model Output**: Final weights, biases, and settings saved to a text file. Error trends over training epochs saved as a plot.
- **Prediction**: Functions implemented for both BGD and SGD models to make predictions for XOR inputs.

All model parameters, including final weights and errors, are saved in `NNModelParameters.txt`. Error trends are visualized in `cost_error_vs_plot.png`.

PSEUDOCODE

1. Initialize

- Set learning rate and number of training iterations
- Define sigmoid activation function and its derivative
- Create XOR dataset with input pairs (X) and expected outputs (y)
- Randomly initialize weights: w1 to w6
- Randomly initialize biases: b1 to b3

==========

2. Batch Gradient Descent Training For each epoch in total_epochs:

- Set total_error = 0
- Reset gradients for all weights and biases

For each (i1, i2), expected_output in dataset: - Feedforward: - h1_input = i1*w1 + i2*w3 + b1 - h1_output = sigmoid(h1_input) - h2_input = i1*w2 + i2*w4 + b2 - h2_output = sigmoid(h2_input) - o1_input = h1_output*w5 + h2_output*w6 + b3 - o1_output = sigmoid(o1_input)

- Compute error = o1_output - expected_output

- Accumulate squared error

==========

- Backpropagation:

- d_o1 = error * sigmoid_derivative(o1_output)

- d_h1 = d_o1 * w5 * sigmoid_derivative(h1_output)

- d_h2 = d_o1 * w6 * sigmoid_derivative(h2_output)

- Accumulate gradients for weights and biases

- Update weights and biases using averaged gradients and learning rate
- Record average error for the epoch

==========

3. Stochastic Gradient Descent Training Re-initialize weights and biases

For each epoch in total_epochs:

- Set total_error = 0

For each (i1, i2), expected_output in dataset: - Perform same feedforward steps - Compute error and deltas (as in batch training) - Update weights and biases immediately - Accumulate squared error

- Record average error for the epoch

==========

4. Save Model Parameters

- Write learning rate, iteration count, final errors, and weights/biases to "NNModelParameters.txt"

==========

5. Plot Error Trend

- Plot error vs iteration for both Batch and SGD and save as "error_plot.png"

==========

6. Predict Function Define predict(i1, i2):

- h1_output = sigmoid(i1*w1 + i2*w3 + b1)
- h2_output = sigmoid(i1*w2 + i2*w4 + b2)
- o1_output = sigmoid(h1_output*w5 + h2_output*w6 + b3)
- Return rounded o1_output

Define predict_sgd(i1, i2):

- Same structure, using SGD-trained weights and biases
- Return rounded o1_output

==========

Parameter	**Value**
Learning Rate	0.5
Epochs	10000
Hidden Nodes	2
Activation	Sigmoid
Weight Initialization	Random (0 to 1)

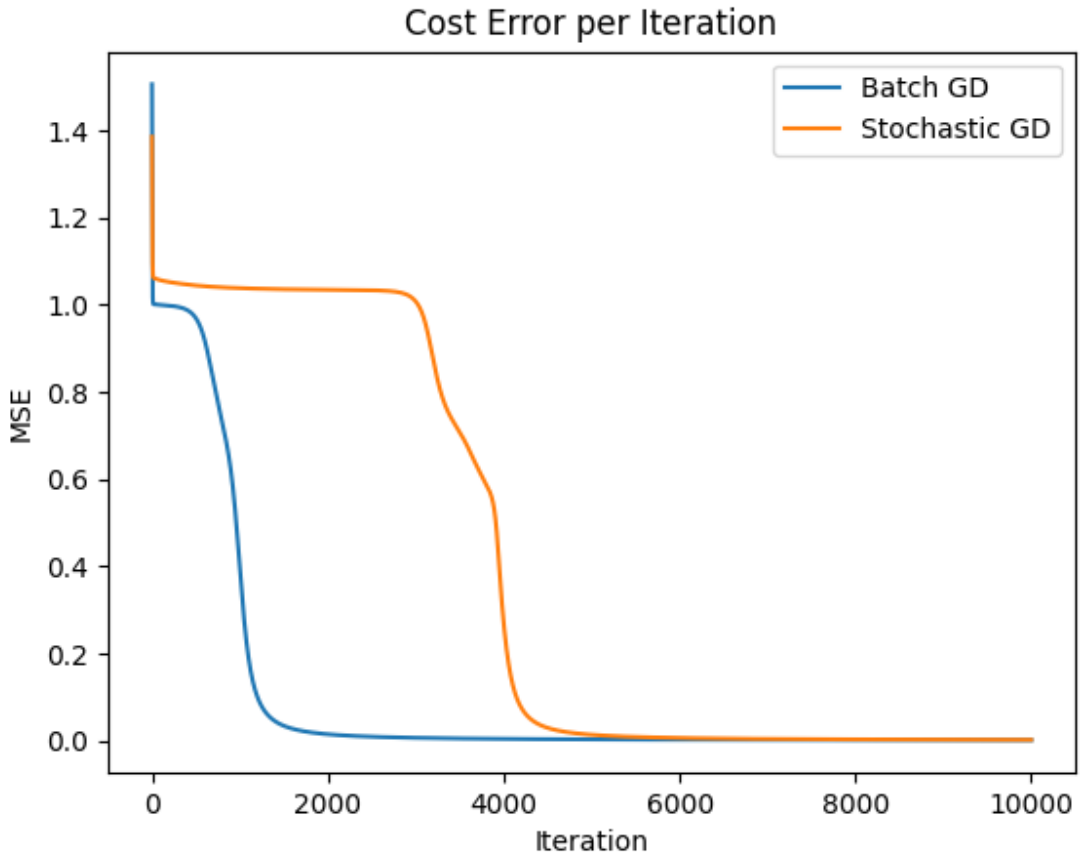
==========

Name: Issa Tijani | Class: Data 527 | Term: Spring 2025 | Due Date: 4/3/25

RESULT

Metric	Batch Gradient Descent	Stochastic Gradient Descent
Learning Rate	0.5	0.5
Iterations	10,000	10,000
Final MSE	0.00122	0.00186
Prediction [0, 0]	0.019	0.019
Prediction [0, 1]	0.983	0.98
Prediction [1, 0]	0.983	0.979
Prediction [1, 1]	0.017	0.026
Weight Range	+4.5 to +10	-6.6 to +9.4
Output Stability	More stable	Slightly noisier
Convergence Curve	Smooth and early	Noisier but effective

Visual Evidence (Error Plot):



Performance Comparison: Batch vs Stochastic Gradient Descent

The XOR neural network was trained using both Batch Gradient Descent (BGD) and Stochastic Gradient Descent (SGD), each for 10,000 iterations with a learning rate of 0.5. Both methods performed well and learned the XOR function, but BGD had a slightly lower final error (0.00122) compared to SGD (0.00186).

BGD's predictions were marginally closer to the ideal outputs (0 or 1) and showed smoother convergence, as reflected in the error plot. SGD, while slightly noisier, still achieved comparable results and offered the advantage of faster, per-sample updates.

In conclusion, BGD offered better precision and stability, while SGD proved to be an efficient alternative with nearly equal performance.

DISCUSSION

The comparative performance of BGD and SGD in solving the XOR problem provides useful insights into their practical application. Despite both algorithms achieving perfect classification accuracy, BGD demonstrated more stable learning behavior and finer precision. This can be attributed to the batch update strategy, which tends to smooth out fluctuations in the error surface.

SGD, while slightly less precise, proved to be an efficient learning approaches especially notable for its real-time updates, which are beneficial when working with large-scale data. The observed difference in weight magnitudes further reinforces the stochastic nature of SGD, where rapid updates can result in more extreme weight adjustments.

In summary, the results reaffirm the suitability of BGD for small, well-defined problems like XOR, while also highlighting SGD's ability to generalize well and remain competitive in performance.

CONCLUSION

This exercise successfully demonstrated the implementation and training of a feedforward neural network to solve the XOR logic problem using both Batch and Stochastic Gradient Descent methods. The network, designed with a 2-2-1 architecture and sigmoid activation functions, effectively classified all XOR inputs with 100% accuracy.

The comparison revealed that while both BGD and SGD achieved strong results, BGD provided more stable learning and slightly better final predictions. SGD, however, maintained its value as a faster and more flexible training method. These findings highlight important trade-offs in neural network optimization strategies and serve as a strong foundation for understanding more complex architectures and real-world datasets.

REFERENCES

- Python Documentation: numpy and matplotlib libraries (https://numpy.org, https://matplotlib.org).
-
- Dataset: XOR Truth Table
- Course Lecture Notes on Neural Networks and Backpropagation
- Online Visual Guide to Gradient Descent: https://en.wikipedia.org/wiki/Gradient_descent
- Nielsen, M. (2015). "Neural Networks and Deep Learning"