Exploring Sparse Linear Algebra in 3D GNN for Scalar Field Prediction

Zicheng Zhao

March 25, 2025

Abstract

This project investigates sparse linear algebra in a 3D Graph Neural Network (GNN) to predict the scalar field, where $f(x, y, z) = x^2 + y^2 + z^2 \approx 1$ on a 500-point unit sphere is used as the instance for implementation here. A Graph Convolutional Network (GCN) leverages a k-NN graph with a sparse adjacency matrix, where we observe its performance from mean squared errors in different precisions float32 and float 64. The study quantifies numerical precision effects and analyzes graph properties (single connected component). Results highlight the role of sparse linear algebra in GNN efficiency, with future applications in 3D mesh processing.

1 Introduction

Graph Neural Networks (GNNs) excel at processing irregular data, such as 3D point clouds, by modeling relationships as graphs and performing computations via linear algebra operations [Kipf and Welling, 2017]. In 3D applications, GNNs are valuable for tasks like scalar field prediction, where sparse linear algebra plays a critical role in computational efficiency. This project explores sparse linear algebra in a 3D GNN to predict the scalar field $f(x, y, z) = x^2 + y^2 + z^2$, which equals 1 on the unit sphere ($x^2 + y^2 + z^2 = 1$), using a 500-point cloud. It also examines numerical precision and graph properties, emphasizing linear algebra's impact on GNN performance.

The GNN, specifically, a graph convolutional network (GCN), with nodes as points, coordinates as features, and edges as neighbor connections, is defined by a sparse adjacency matrix A. Sparse linear algebra underpins the GCN's efficiency, as operations like matrix multiplications exploit the graph's sparsity to reduce computational complexity. The GCN is trained to minimize the mean squared error (MSE) over 100 epochs, with numerical precision (float32 vs. float64) analyzed for its effect on training and inference. Graph properties, such as connectivity and degree distribution, are assessed to understand their influence on GNN propagation.

The objectives are:

i Accurate scalar prediction using a GCN.

ii Quantification of numerical precision effects through sparse matrix operations.

iii Analysis of graph properties via linear algebra, with a focus on sparsity.

The results demonstrate the GCN's effectiveness, highlight precision impacts, and underscore the importance of sparse linear algebra in 3D GNN applications, with potential extensions to mesh processing in tools like Blender.

2 Mathematical Foundations

This section details the linear algebra foundations of the project, focusing on sparse matrices, graph theory, and GCN computations.

2.1 Graph Theory and the Laplacian

The 500-point cloud is modeled as an undirected graph $G = (V, E)$, where G is the graph, V is the set of nodes (points), and E is the set of edges (k-NN connections). The adjacency matrix $A \in \mathbb{R}^{n \times n}$ is:

$$A_{ij} = \begin{cases} 1 & \text{if nodes } i \text{ and } j \text{ are neighbors} \\ 0 & \text{otherwise} \end{cases}$$

The degree matrix $D \in \mathbb{R}^{n \times n}$ is diagonal, with $D_{ii} = \sum_j A_{ij}$, the degree of node i.

The combinatorial Laplacian is:

$$L = D - A$$

The Laplacian matrix $L \in \mathbb{R}^{n \times n}$ is:

$$L_{ij} = \begin{cases} deg(v_i) & \text{if } i = j, \text{ where } deg(v_i) \text{ represents the degree of node i} \\ -1 & \text{if } i \neq j \text{ and node } i \text{ and } j \text{ are neighbors} \\ 0 & \text{otherwise} \end{cases}$$

For example, a small graph with 4 nodes forming a path (1-2-3-4):

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad L = D - A = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}.$$

The Laplacian's eigenvalues indicate connectivity: a single 0 eigenvalue ensures a connected graph, facilitating GNN propagation.

2.2 Sparse Linear Algebra in GNNs

The adjacency matrix A is sparse, with non-zero entries (edges).

$$\text{Sparsity} = 1 - \frac{\text{nnz}}{n^2}, \text{ where nnz represents the number of non-zero elements}$$

Sparse matrices reduce the complexity of operations like matrix-vector multiplication from $O(n^2)$ to $O(\text{nnz})$, critical for GNN scalability.

For example, computing $A\mathbf{v}$ for a vector $\mathbf{v} \in \mathbb{R}^{500}$ and an adjacency matrix A with 4500 non-zero entries (edges) involves only 4,500 multiplications, not 250,000.

2.3 GCN Convolution

The GCN in this project performs convolutions on the graph to predict the scalar field f. To understand GCNs, consider how convolutions work in traditional CNNs: they aggregate information from nearby pixels using a filter. On a graph, there's no grid, so we use the graph's structure—nodes (points) and edges (connections)—to define "nearby." Each node aggregates features from its neighbors, much like a message-passing process [Kipf and Welling, 2017].

First, self-loops are added to the adjacency matrix:

$$\tilde{A} = A + I,$$

where I is the identity matrix ($I_{ii} = 1$, $I_{ij} = 0$ for $i \neq j$). Self-loops ensure that each node includes its own features during aggregation, which is standard in GCNs to capture local information and stabilize training [Kipf and Welling, 2017].

The degree matrix of $\tilde{A}$ is:

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij},$$

so $\tilde{D}_{ii} = D_{ii} + 1$. The GCN convolution is:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}\right),$$

where l represent the number of layers, $H^{(l)} \in \mathbb{R}^{n \times d_l}$ is the feature matrix, $W^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}$ is the weight matrix, where n represents the number of nodes and d represents the number of

input features in nodes, and σ is ReLU activation function, ensures the nonlinearity.

This operation aggregates features from a node's neighbors (including itself), with $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ normalizing contributions to balance nodes with different degrees. For example, if a node has degree 20 and its neighbor has degree 10, their contributions are scaled by $1/\sqrt{\tilde{D}_{ii}}$ and $1/\sqrt{\tilde{D}_{jj}}$, ensuring fairness. The reason why $D^{-1}A$ isn't used instead is explained later.

In this project, for the sake of simplification, the GCN has two layers:

- Layer 1: $H^{(0)} \in \mathbb{R}^{500\times 3}$, where represents 500 points in 3D coordinates, is transformed to $H^{(1)} \in \mathbb{R}^{500\times 16}$, using $W^{(0)} \in \mathbb{R}^{3\times 16}$.
- Layer 2: $H^{(1)}$ is transformed to $H^{(2)} \in \mathbb{R}^{500\times 1}$, using $W^{(1)} \in \mathbb{R}^{16\times 1}$, predicting f.

The convolution is inspired by spectral graph theory but simplified for efficiency. Instead of computing eigenvalues of the Laplacian, the GCN uses this normalized form, which approximates a spectral convolution while exploiting the sparsity of $\tilde{A}$ for fast computation.

An alternative normalization, such as $D^{-1}A$, lacks symmetry and self-loops. For a 3-node path graph (1-2-3):

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

we compute $\tilde{A} = A + I$:

$$\tilde{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \quad \tilde{D} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}, \quad \tilde{D}^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{3}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix},$$

yielding:

$$\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2} = \begin{bmatrix} \frac{1}{2} & \frac{1}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{3} & \frac{1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{6}} & \frac{1}{2} \end{bmatrix}.$$

In contrast, $D^{-1}A$:

$$D^{-1}A = \begin{bmatrix} 0 & 1 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \end{bmatrix},$$

is not symmetric and lacks self-loops, leading to imbalanced contributions and potential numerical instability.

3 Methods

3.1 Data Preparation

The dataset was generated as follows:

1. Sample 500 points on the unit sphere using spherical coordinates:

$$x = \sin\theta\cos\phi, \quad y = \sin\theta\sin\phi, \quad z = \cos\theta,$$

   where $\theta \sim \text{Uniform}(0, \pi)$, $\phi \sim \text{Uniform}(0, 2\pi)$.

2. Compute the ground truth $f = x^2 + y^2 + z^2 \approx 1$, adding noise $\mathcal{N}(0, 0.1)$.

3. Construct the k-NN graph ($k = 8$):

   - Compute pairwise Euclidean distances between all 500 points.
   - For each point, select its 8 nearest neighbors based on Euclidean distance, forming directed edges.
   - Ensure the graph is undirected by adding both (i, j) and (j, i) whenever either direction exists. This step accounts for asymmetry in k-NN selection: if node i selects node j as a neighbor, node j may not select node i (e.g., if i is j's 9th nearest neighbor). Adding both directions increases the number of edges and degrees beyond the expected $k = 8$.
   - If the k-NN graph were perfectly symmetric (i.e., i picks j if and only if j picks i), the number of undirected edges would be $\frac{n \times k}{2} = \frac{500 \times 8}{2} = 2000$. However, due to asymmetry, the actual number of edges is expected to be higher, consistent with the degrees ranging from 8 to 14 (from the running result). Using the handshaking lemma ($\sum_i deg(i) = 2E$), the total degree sum and average degree will be determined after rerunning the code.
   - Store the edges as a 2×4748 tensor in PyTorch Geometric (from the running result), where the first row contains source node indices and the second row contains target node indices. Since the graph is undirected, each edge (i, j) is stored twice ((i, j) and (j, i)) for message passing, resulting in 2×2374 directed edges.

3.2 GCN Architecture and Design Rationale

The GCN architecture was:

- **Input:** $H^{(0)} \in \mathbb{R}^{500 \times 3}$, the 3D coordinates.
- **Layer 1:** $H^{(1)} = \text{ReLU}\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(0)}W^{(0)}\right)$, $W^{(0)} \in \mathbb{R}^{3 \times 16}$, $H^{(1)} \in \mathbb{R}^{500 \times 16}$.
- **Layer 2:** $H^{(2)} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(1)}W^{(1)}$, $W^{(1)} \in \mathbb{R}^{16 \times 1}$, $H^{(2)} \in \mathbb{R}^{500 \times 1}$.

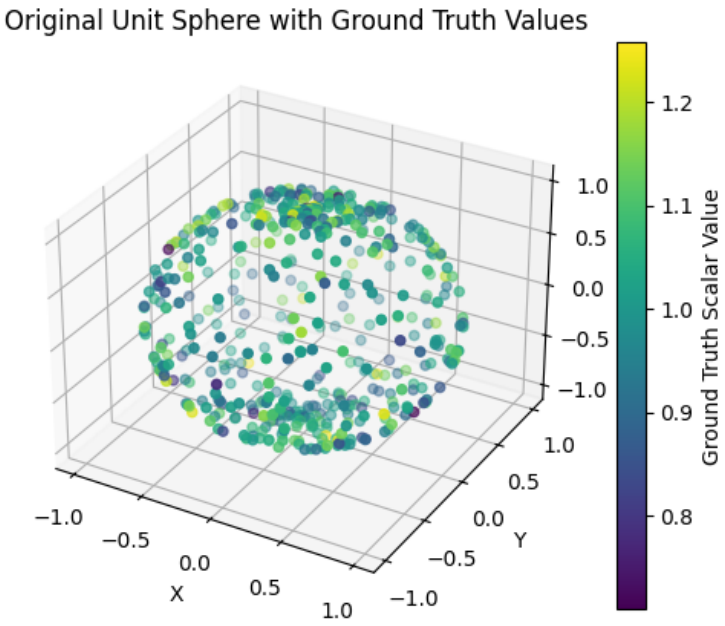

Figure 1: Original Point Cloud, 0.71037-1.25797

- **Output:** Predicted scalars.

The 2-layer design balances expressivity and efficiency, with 16 hidden features capturing local patterns. ReLU adds non-linearity, and the output layer omits activation for regression.

3.3 Training and Loss

Training minimized MSE (Mean Squared Error):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

using the Adam optimizer (learning rate = 0.01) for 100 epochs. Adam (Adaptive Moment Estimation) combines the benefits of momentum and RMSProp by maintaining exponentially decaying averages of past gradients (first moment, m_t) and squared gradients (second moment, v_t) to adaptively adjust the learning rate for each parameter [Kingma and Ba, 2014]. The update rule for a parameter θ_t (e.g. weights $W^{(0)}$, $W^{(1)}$) at step t is:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial \text{MSE}}{\partial \theta_{t-1}}, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial \text{MSE}}{\partial \theta_{t-1}}\right)^2,$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \quad \theta_t = \theta_{t-1} - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon},$$

where $\eta = 0.01$ is the learning rate, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are the default exponential decay rates for the first and second moments, and $\epsilon = 10^{-8}$ prevents division by zero.

This is not the focus of the project, while it is still worthwhile to mention that Adam's adaptive learning rate helps the GCN to converge efficiently on the sparse graph, balancing the trade-off between speed and stability during training over 100 epochs.

3.4 Precision Analysis

Precision analysis involved:

1. Train two GCNs (float32, float64), compute MSE.
2. Compute MAE (Mean Absolute Error) between predictions:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i^{\text{float32}} - y_i^{\text{float64}}|.$$

3. Compute rounding error: Forward pass the float64-trained model parameters in both float64 and float32 precisions, and compute the mean absolute difference between the two sets of predictions. This measures the inference-time precision loss of float32 relative to float64 for the float64-trained model.

3.5 Graph Analysis

- **Sparsity:** $1 - 2374/250000 = 0.9905$.
- **Degrees:** Compute D_{ii}, range 8-14, reflecting the undirected k-NN construction.
- **Connectivity:** Use `scipy.sparse.csgraph.connected_components` to confirm.

4 Results

The GCN predicted $f \approx 1$, with MSE 0.0153041556 (float32) and 0.0145649973 (float64), MAE 0.0476326102, and rounding error 0.0000000494. The k-NN graph ($k = 8$) had 2374 undirected edges (stored as 4748 directed edges in a 2×4748 tensor), with sparsity 0.9905, degrees 8-14, and 1 component.

Figure 2 shows the single precision predictions (0.84640-1.21998), while Figure 3 shows the double precision predictions (0.85257-1.28046). [Obtained from training]

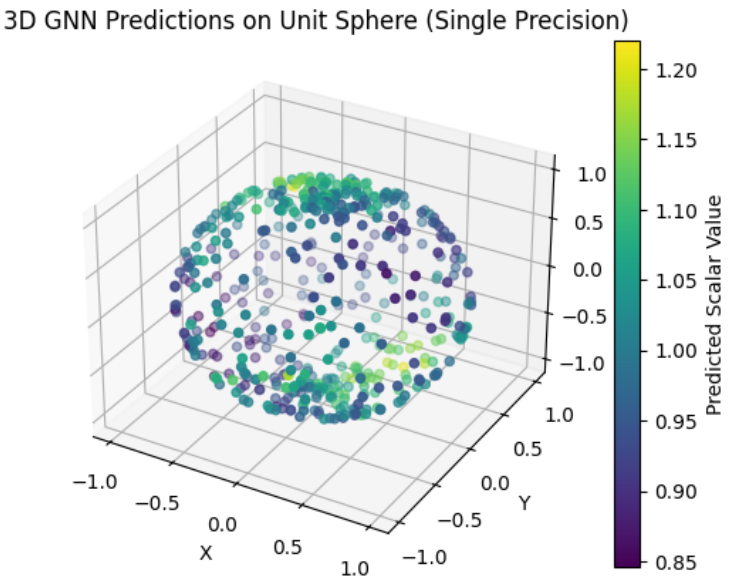

Figure 2: 3D GNN Predictions on Unit Sphere (Single Precision, 0.84640-1.21998)

5 Discussion

The GCN's MSEs' difference validates scalar prediction, driven by sparse matrix operations like $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$. Sparsity (0.9905) reduces complexity, as $\tilde{A}$ only has 2374 non-zero elements, making operations efficient. Connectivity (1 component) ensures propagation, with degrees (8-14) supporting robust information flow.

The ground truth values (0.71037-1.25797, see Figure 1) provide a baseline for comparison. The prediction ranges are narrower than the ground truth, indicating that the GCN smooths out some of the variability introduced by the noise, which is expected for a regression model minimizing MSE. The double precision model's wider range extends slightly beyond the ground truth maximum, suggesting it captures more of the upper variability, while the single precision model's upper bound is more conservative. The lower bounds of both predictions are higher than the ground truth minimum, indicating that the GCN struggles to predict the lower extremes, possibly due to the smoothing effect of graph convolution.

MAE shows precision impacts training, likely due to gradient accumulation, while the rounding error confirms float32's inference stability. The rounding error, computed as the mean absolute difference between float64 and float32 predictions of the float64-trained model, reflects the precision loss due to float32's limited mantissa (23 bits vs. 52 bits in float64), which truncates small numerical differences that float64 can represent. This small error indicates that sparse operations in the GCN (e.g., matrix multiplications with $\tilde{A}$) minimize error propagation during inference, making float32 a viable choice for deployment despite its lower precision. However, the higher MSE in float32 suggests that training-time precision

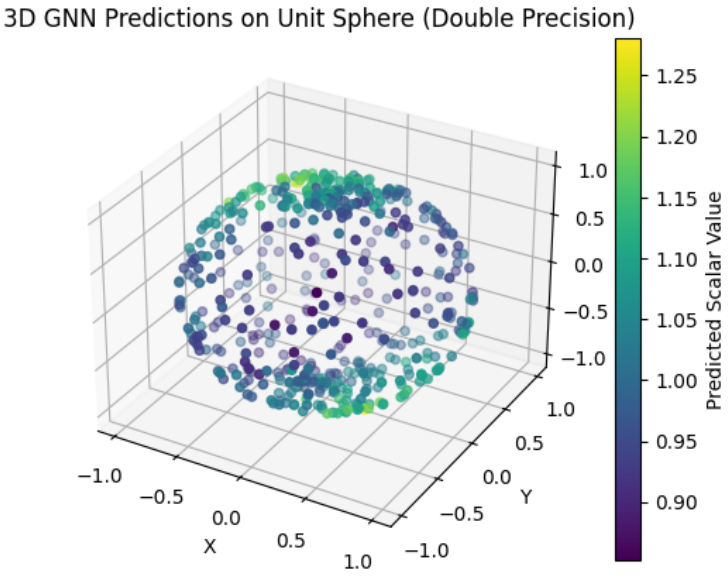

Figure 3: 3D GNN Predictions on Unit Sphere (Double Precision, 0.85257-1.28046)

differences have a more significant impact, as float32's lower precision leads to accumulated errors in gradient updates over 100 epochs, resulting in a less optimal model. Double precision's edge (0.0007391583 MSE difference) aids training, but single precision suffices for inference, informing hardware choices. The double precision model's ability to capture a wider range aligns with its lower MSE, suggesting better numerical stability. This highlights sparse linear algebra's role in 3D GNNs, with applications in mesh processing.

For a comparison of the training dynamics, see Appendix A, the plot of training losses.

6 Conclusion and Future Work

For the section that involves self-loop, [Chen et al., 2023] highlights a counter-intuitive effect: in GNNs with two or more layers, self-loops can reduce the information a node retains about itself, particularly in random graphs with arbitrary degree sequences. Their experiments on synthetic graphs with noisy topologies suggest this may impact tasks like node classification. In this project, the 2-layer GCN uses self-loops, but this finding warrants future exploration for deeper GNNs.

The achieved MSEs by GCN leveraged sparse linear algebra for scalar prediction. MAE and rounding error quantified precision effects, while sparsity and connectivity validated efficiency. Future work will explore sparse linear algebra in larger 3D datasets and mesh processing in platforms like Blender, potentially revisiting the role of self-loops in light of recent findings on their impact in multi-layer GNNs.

A Training Loss Comparison

Figure 4 shows the training loss (MSE) for the float32 and float64 models over 100 epochs. The float64 model consistently achieves a slightly lower loss, reflecting its better numerical stability and convergence, as evidenced by the final MSE values.

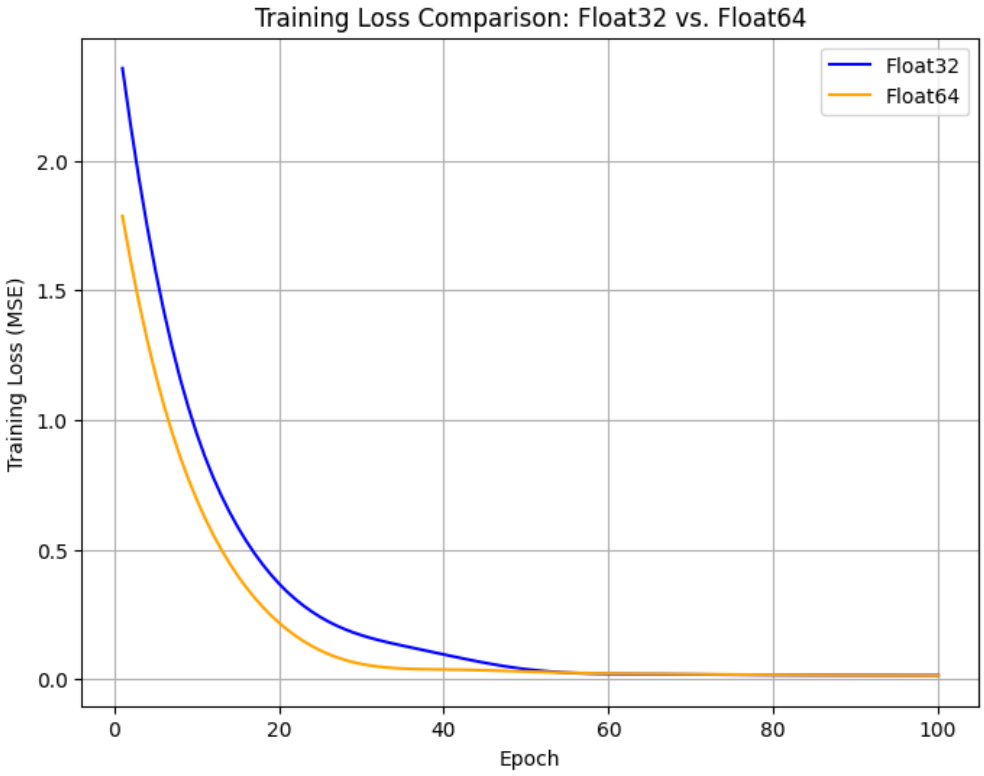

Figure 4: Training Loss (MSE) for Float32 and Float64 Models over 100 Epochs

References

Derek Lim Chen, Felix Yu, and Andrew M. Saxe. The surprising counter-intuitiveness of adding self-loops in graph neural networks. *arXiv preprint arXiv:2312.01721*, 2023.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. URL `https://arxiv.org/abs/1412.6980`.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2017.