# DSA5208: Project 1 Report

Tong
Lyu S
Zhou H

October 4, 2025

## 1 Abstract

In this project, we presents the implementation of a distributed Stochastic Gradient Descent (SGD) framework using the Message Passing Interface (MPI) to train a neural network on the 2022 New York City Taxi dataset. Several refinements were adopted to improve the whole job: During data preprocessing, the categorical features were represented through embedding techniques instead of traditional one-hot encoding, which significantly reducing dimensionality and enhancing model generalization. What's more, in model construction and training stage, we tried different kinds of combinations of hyperparameters and utilized a grid-search method to find the best one and we refined the early stopping mechanism to make training process more robust. Additionally, we investigated several MPI schemes for RMSE calculation which improved the computing efficiency. Our experiments show the proposed system achieves efficient parallel training across multiple nodes, demonstrating improved convergence speed and predictive performance.

## 2 Theoretical Framework

### 2.1 Shallow Neural Network Model

Our model for this regression task is a shallow neural network with a single hidden layer. A key challenge in our dataset is the presence of both numerical and high-cardinality categorical features. To handle this effectively, we integrated **embedding layers** to process the categorical data.

**Categorical Feature Embeddings** In our project, we avoided one-hot encoding for categorical variables such as `PULocationID` (with over 260 unique values), because it creates extremely high-dimensional, sparse feature vectors. This approach is inefficient in terms of memory and can hinder model training. Instead, we adopted the embedding technique, where each category is mapped to a low-dimensional, dense, and learnable vector.

Specifically, for a categorical feature with a vocabulary of size $V$, we define a learnable embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times d_{emb}}$, where the embedding dimension $d_{emb} \ll V$. This matrix acts as a lookup table: a category's integer index $i$ retrieves the corresponding vector $\mathbf{e}_i$ (the $i$-th row of $\mathbf{E}$). This vector is then treated as part of the model's input. This technique significantly reduces input dimensionality and allows the model to learn semantic relationships between categories, leading to a notable improvement in scalability and model performance.

**Model Parameters** The complete set of learnable parameters, $\theta$, for our model includes:

- **Embedding Matrices**: A set of matrices $\{\mathbf{E}_{cat}\}$ for each categorical feature.

- **Hidden Layer Weights**: $\mathbf{W} \in \mathbb{R}^{n \times m_{combined}}$
- **Hidden Layer Biases**: $\mathbf{b} \in \mathbb{R}^n$
- **Output Layer Weights**: $\mathbf{v} \in \mathbb{R}^{1 \times n}$
- **Output Layer Bias**: $b_{out} \in \mathbb{R}$

Here, $n$ is the number of hidden neurons, and $m_{combined}$ is the total dimension of the input vector after concatenating numerical features with all their corresponding embedding vectors.

**Forward Propagation** Given a single sample with numerical features $\mathbf{x}_{numeric}$ and categorical feature indices $\{i_{cat}\}$, the model computes a prediction $\hat{y}$ via the following forward pass:

$$\begin{aligned}
\mathbf{e}_{cat} &= \mathbf{E}_{cat}[i_{cat}] && \text{(1. Embedding lookup)} && (1)\\
\mathbf{x}_{combined} &= \text{concat}(\mathbf{x}_{numeric}, \mathbf{e}_{pu}, \mathbf{e}_{do}, \dots) && \text{(2. Concatenate all features)} && (2)\\
\mathbf{z} &= \mathbf{W}\mathbf{x}_{combined} + \mathbf{b} && \text{(3. Hidden layer pre-activation)} && (3)\\
\mathbf{a} &= \sigma(\mathbf{z}) && \text{(4. Hidden layer activation)} && (4)\\
\hat{y} &= \mathbf{v}\mathbf{a} + b_{out} && \text{(5. Final prediction)} && (5)
\end{aligned}$$

where $\sigma(\cdot)$ represents a non-linear activation function such as Sigmoid, Tanh, or ReLU.

**Loss Function** For a single training sample, we use the Mean Squared Error (MSE) loss, scaled by $\frac{1}{2}$ to simplify the gradient calculation during backpropagation:

$$L(\theta) = \frac{1}{2}(\hat{y} - y)^2 \tag{6}$$

where $y$ is the true target value.

## 2.2 Gradient Calculation via Backpropagation

We compute the gradient of the loss function with respect to all parameters in $\theta$ using the chain rule, in a process known as backpropagation. This process starts from the output layer and moves backward through the network.

**Step 1: Output Layer Gradients** First, we compute the gradient of the loss with respect to the model's prediction $\hat{y}$, which is the initial error signal:

$$\frac{\partial L}{\partial \hat{y}} = (\hat{y} - y) \tag{7}$$

Using this, we find the gradients for the output layer parameters, $b_{out}$ and $\mathbf{v}$. Applying the chain rule:

$$\frac{\partial L}{\partial b_{out}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_{out}} = (\hat{y} - y) \cdot 1 = (\hat{y} - y) \tag{8}$$

$$\frac{\partial L}{\partial \mathbf{v}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{v}} = (\hat{y} - y)\mathbf{a}^T \tag{9}$$

**Step 2: Hidden Layer Gradients** Next, we propagate the error backward to the hidden layer. To find the gradients for $\mathbf{W}$ and $\mathbf{b}$, we first need to compute the error signal $\delta_h$, defined as the gradient of the loss with respect to the hidden layer's pre-activation $\mathbf{z}$.

$$\begin{aligned}\delta_h \equiv \frac{\partial L}{\partial \mathbf{z}} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \\ &= (\hat{y} - y) \cdot \mathbf{v}^T \odot \sigma'(\mathbf{z}) \end{aligned} \tag{10}$$

where $\odot$ denotes the element-wise product, and $\sigma'(\mathbf{z})$ is the derivative of the activation function. With this error signal, we can easily compute the gradients for the hidden layer's weights and biases:

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{b}} = \delta_h \odot \mathbf{1} = \delta_h \tag{11}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \delta_h \mathbf{x}_{combined}^T \tag{12}$$

The last step is an outer product between the error signal vector $\delta_h$ and the combined input vector $\mathbf{x}_{combined}$.

**Step 3: Embedding Layer Gradients** Finally, we propagate the gradient back to the input layer to update the embedding matrices. We first calculate the gradient of the loss with respect to the combined input vector $\mathbf{x}_{combined}$:

$$\frac{\partial L}{\partial \mathbf{x}_{combined}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{x}_{combined}} = \mathbf{W}^T \delta_h \tag{13}$$

This resulting gradient vector, $\frac{\partial L}{\partial \mathbf{x}_{combined}}$, has the same dimension as $\mathbf{x}_{combined}$ and contains the gradients corresponding to both the numerical features and the embedding vectors.

Crucially, the gradient for an entire embedding matrix $\mathbf{E}_{cat}$ is sparse. The only non-zero gradient is for the specific embedding vector $\mathbf{e}_{cat}$ that was retrieved during the forward pass (i.e., the $i_{cat}$-th row). The portion of the gradient vector $\frac{\partial L}{\partial \mathbf{x}_{combined}}$ that corresponds to $\mathbf{e}_{cat}$ is used to update this specific row. All other rows of the embedding matrix $\mathbf{E}_{cat}$ have a gradient of zero for this sample. This allows for an efficient, sparse update of the embedding parameters.

### 2.3 Parameter Update with Stochastic Gradient Descent

We update the parameters using Mini-batch Stochastic Gradient Descent (SGD). After computing the gradients for a mini-batch of $M$ samples, the update rule for any parameter $\phi \in \theta$ (where $\phi$ could be $\mathbf{W}, \mathbf{b}, \mathbf{v}, b_{out}$, or any $\mathbf{E}_{cat}$) is:

$$\phi_{k+1} = \phi_k - \eta \frac{1}{M} \sum_{i=1}^{M} \nabla_\phi L_i(\phi_k) \tag{14}$$

where $\eta$ is the learning rate and $k$ is the iteration step.

## 3 Experimental Design and Methodology

### 3.1 Environment Setup

**Hardware Architecture** We built our experimental environment on the Google Cloud Platform (GCP).Our cluster consisted of two identical Virtual Machine (VM) nodes, explicitly named node1 and node2. Each node was configured as a c2-standard-8 instance, providing 8 vCPUs and 32GB of RAM.

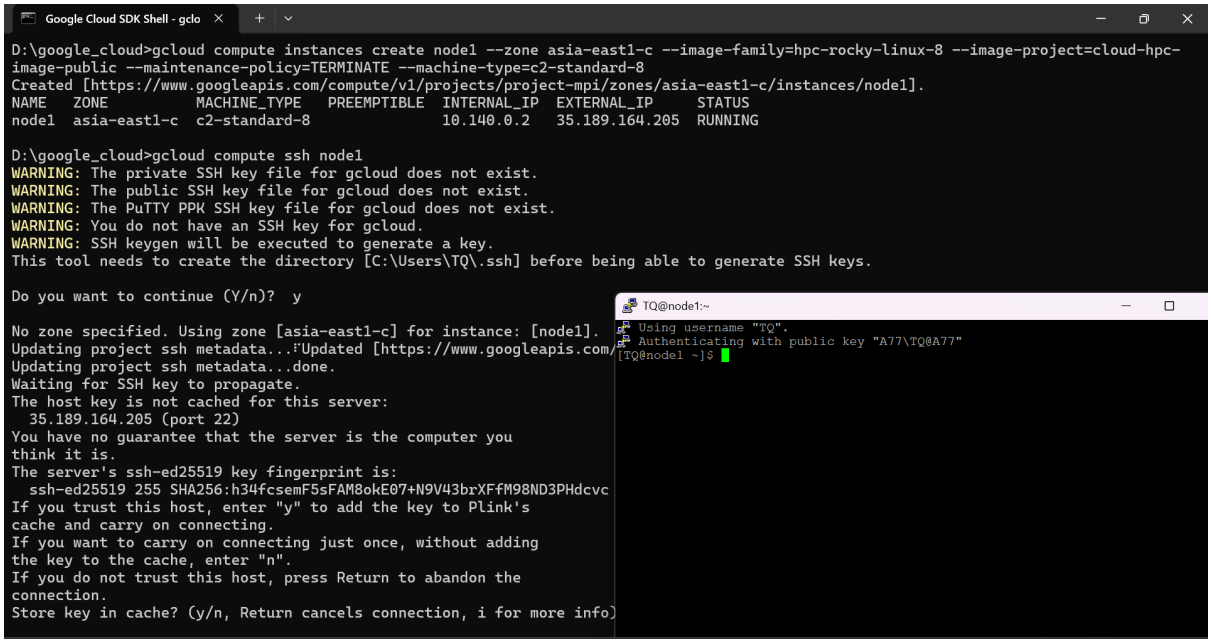



Figure 1: VM Instance Creation via Google Cloud Shell.

This memory configuration was crucial for our Master-Slave data distribution strategy: the Rank 0 process (located on node1) was responsible for loading the entire 7 million-row dataset. The 32GB RAM ensured that node1 could successfully read and cache the complete dataset without incurring memory overhead or bottlenecks during the initial data scattering phase.

We established passwordless SSH between the nodes (i.e., node1 could SSH into node2 without a prompt), which is a prerequisite for the mpiexec command to launch processes across the cluster.

**Software Stack** To ensure our experiments were reproducible, we installed the same software stack on each node:

- **MPI Library:** Intel MPI 2021
- **Programming Language:** Python 3
- **Core Libraries:** NumPy, Joblib, Matplotlib
- **MPI Bindings:** mpi4py

### 3.2 Data Preprocessing

Our preprocessing pipeline involved several steps to ensure data quality and prepare it for the model. First, we dropped rows with missing critical values and filtered out trips with non-positive fares. From pickup and dropoff times, we engineered new features (`trip_duration`, `pickup_hour`), and applied a log-transform to the skewed target variable (`total_amount`) for stability. Categorical features were converted into compact integer indices suitable for embedding layers.

The data was then split into training (70%) and testing (30%) sets and we saved all preprocessed arrays along with the scaler, categorical vocabularies, and metadata. This ensures the dataset is both model-ready and fully reproducible for distributed training.

### 3.3 Model Design

**Initial Challenge and Design Choice** Our primary objective was to develop an effective regression model for predicting continuous target values in the NYC Taxi dataset. We initially adopted a shallow neural network architecture due to its simplicity and interpretability. However, the key challenge emerged from the coexistence of numerical and high-cardinality categorical features—most notably `PULocationID`, which contains over 260 unique identifiers.

First, we tried feeding the raw integer indices of categorical features directly into the network as numeric inputs. This is problematic because it forces the model to interpret an arbitrary ordering and scale between categories, which misrepresents their discrete, non-ordinal nature and can induce spurious numeric relationships. Next, we considered one-hot encoding. Although one-hot preserves the categorical identity without imposing order, it becomes impractical for high-cardinality features: it produces extremely sparse, very high-dimensional inputs, greatly increases memory and computational cost, and exacerbates the curse of dimensionality—leading to slower convergence and poorer scalability.

To address this limitation, we incorporated embedding layers into the network. Inspired by techniques in Natural Language Processing, this design treats each categorical ID as a "token" and learns a dense, low-dimensional vector representation during training. These embeddings not only reduce input dimensionality but also enable the model to capture latent correlations among categorical entities—such as geographic proximity between pickup and drop-off locations.

Table 1 compares model performance with and without embeddings under an early stopping criterion. The results clearly demonstrate that introducing embedding layers significantly accelerates convergence and improves predictive accuracy.

Table 1: Comparison of Model Performance With and Without Embedding Layers

| Configuration | Convergence Epoch | Final Loss | Train RMSE |
|---|---|---|---|
| Without Embedding | 4 | 154.67 | 20.05 |
| With Embedding | 12 | 1.75 | 1.96 |

From these results, we observe a dramatic reduction in both loss and RMSE values when embedding layers are introduced. The model without embeddings converged within only four epochs but plateaued at a high loss value (154.67), indicating limited learning capability. In contrast, the model with embeddings achieved a much lower final loss (1.75) after twelve iterations, demonstrating its superior ability to represent categorical variables and improve overall regression accuracy.

**Hyperparameter Grid Search** To ensure efficiency during the extensive search across 45 parameter combinations, the hyperparameter tuning was performed on a fixed subsample of approximately **7 million data points** (around 20% of the total dataset). Critically, this entire grid search process was executed on the 2-VM cluster using a total of **8 MPI processes** (4 processes per node) to balance parallel speedup and computational overhead. The final selected configuration was then validated on the complete dataset in subsequent sections.

With the network architecture defined, the next step was to optimize its performance by selecting the best hyperparameters. We conducted a distributed grid search to systematically evaluate different configurations. We chose **Sigmoid, Tanh, and ReLU** as our activation functions because they are the most common and effective choices in deep learning, each offering distinct properties for handling different types of data and preventing issues like the vanishing gradient problem.

Our search focused on three key hyperparameters:

- **Activation Function:** Sigmoid, Tanh, ReLU.

- **Batch Size:** 16, 32, 64, 128, 256.
- **Number of Hidden Neurons:** 16, 32, 64.

To ensure a fair comparison and isolate the impact of these specific parameters, we intentionally held the learning rate constant at $\eta = 0.01$. This value was chosen as a reasonable starting point, balancing the need for the model to learn efficiently without causing instability or overshooting the optimal solution. Each configuration was trained for up to 500 iterations, and the model with the lowest **Root Mean Squared Error (RMSE)** on the test set was selected as our final, optimized model.

**Final Architecture** Our final model architecture, depicted in Figure 2, is a fully-connected network that processes the heterogeneous input data as follows:

1. The 6 scaled numerical features are taken as direct input.
2. The 4 categorical features are fed into their respective embedding layers. We chose embedding dimensions of 16 for the high-cardinality features (`PULocationID`, `DOLocationID`) and 4 for the lower-cardinality ones (`RatecodeID`, `payment_type`).
3. The numerical vectors and the newly created embedding vectors are concatenated into a single, combined feature vector.
4. This vector is passed through a single hidden layer with a tunable number of neurons ($n$) and a non-linear activation function.
5. Finally, a single output neuron produces the final regression prediction.

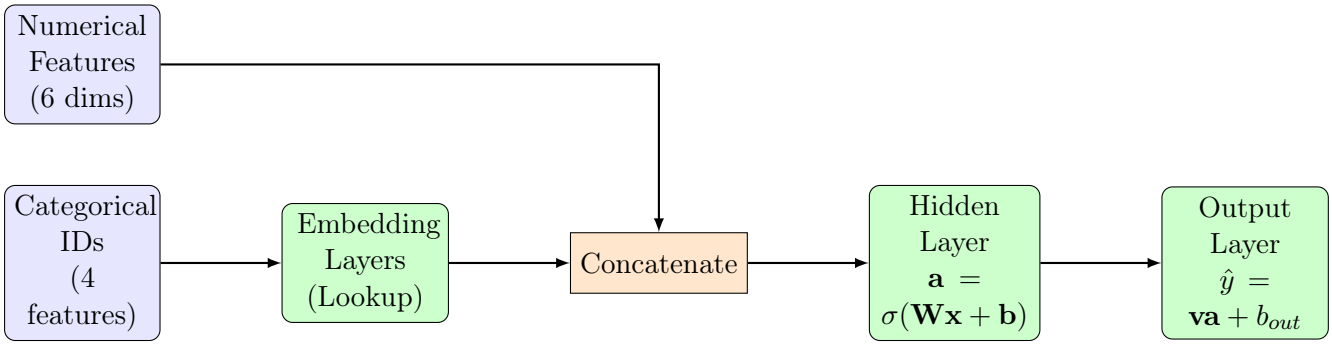



Figure 2: The architecture of our shallow neural network, showing how numerical and categorical features are processed and combined.

## 3.4 Distributed Training Framework with MPI

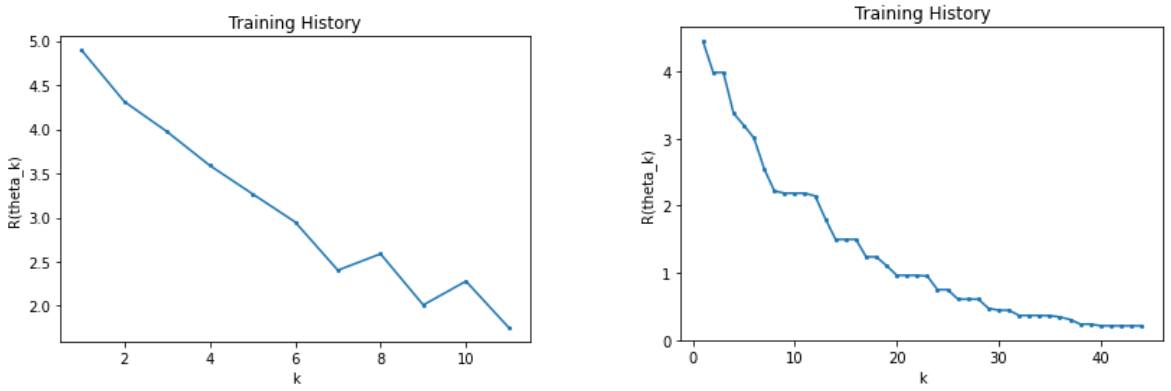
### 3.4.1 Core Implementation: Data Parallelism and Synchronous SGD

To accelerate the computationally intensive grid search, we implemented a distributed training framework using the Message Passing Interface (MPI), following the data parallelism paradigm.

The master process (Rank 0) first loads and partitions the dataset into $N$ disjoint subsets, where $N$ is the total number of MPI processes. Each worker receives one subset through the `MPI.Scatter` operation, while the master releases its copy to conserve memory.

The training procedure adopts a synchronous Stochastic Gradient Descent (SGD) scheme:

1. **Parameter Broadcast:** For each grid search trial, the master process broadcasts the current hyperparameter configuration and the initial model weights ($\theta_0$) to all worker processes.




(a) Early stopping (sensitive)

(b) Early stopping with 3-epoch patience

Figure 3: Effect of Early Stopping Sensitivity on Convergence Stability

2. **Synchronous Gradient Descent:** Each worker independently computes gradients on its local data partition. The local gradients are then aggregated using the `MPI.Allreduce` operation to obtain a global average. This ensures that all processes apply identical parameter updates, keeping the models perfectly synchronized throughout training.

3. **Refined Early Stopping Mechanism:** The early stopping criterion plays a key role in controlling convergence stability and computational cost. Initially, a highly sensitive rule was applied—training stopped immediately once the loss failed to decrease in a single iteration. As shown in Figure 3a, this resulted in premature convergence at the $12^{\text{th}}$ iteration, with a final loss of 1.75 and training RMSE of 1.96.

   To mitigate this over-sensitivity, we refined the early stopping condition by introducing a *patience window* of three iterations. Under this improved mechanism, training only halts if the global loss fails to improve for three consecutive epochs. As shown in Figure 3b, this adjustment significantly enhanced convergence stability: the model continued to train until the $45^{\text{th}}$ iteration, achieving a final loss of 0.21 and a training RMSE of 0.70. This demonstrates that a moderate patience setting allows the network to escape shallow local minima and reach a more optimal solution.

### 3.4.2 Optimization: Efficient RMSE Calculation

A key part of our implementation involved carefully considering the communication patterns for collective operations, especially for calculating the global RMSE after each training trial. The goal was to minimize communication overhead. We analyzed three distinct MPI schemes: the sequential `MPI_Reduce`+`MPI_Bcast`, the integrated `MPI_Allreduce`, and a minimal `MPI_Reduce`-only approach.

Our analysis was driven by two principles: selecting the most efficient MPI primitive and minimizing the communication scope. While `MPI_Allreduce` is generally superior to a manual `MPI_Reduce`+`MPI_Bcast` due to underlying optimizations (like Recursive Doubling), we realized that for the RMSE calculation, only the master process (Rank 0) needs the final result to log performance and control the grid search.

Therefore, we adopted the `MPI_Reduce`, only strategy as our final optimal choice. In this scheme, each worker computes its local sum of squared errors and sample count, and both values are sent to the master via two separate `MPI_Reduce` operations. The master then performs the final RMSE calculation locally. As shown in Table 2, this strategy is optimal for our use case because it represents the minimal necessary communication (many-to-one) and completely avoids the broadcast overhead of sending the final result back to workers who do not need it.

Table 2: Efficiency comparison of MPI schemes for RMSE calculation.

| **Scheme** | **Result Scope** | **Best Use Case** |
|---|---|---|
| Reduce + Bcast | All Processes | All processes require the final, calculated result. |
| Allreduce | All Processes | All processes need the aggregated sum, but not necessarily the final calculated metric. |
| **Final Optimal (Reduce)** | **Only** Rank 0 | **Only the master needs the result (for logging, control flow).** |

The distributed optimization strategy, which facilitates the parallel hyperparameter search and the efficient, synchronized calculation of the global Test RMSE, is formally summarized in Algorithm 1. The core synchronized gradient descent step within this workflow is detailed in Algorithm 2.

**Algorithm 1** Grid Search and rmse calculation using MPI

Initialize MPI communicator *comm*, get process *Rank* and size $N$
**if** $Rank == 0$ (Master process) **then**
  Load and partition dataset into $N$ chunks of *local_ data*
  Create list of all hyperparameter combinations *combos*
**end if**
Scatter data chunks from master to all processes; each receives *local_ data*
Broadcast *combos* from master to all processes
**for** each *hyperparameter_ combo* in *combos* **do**
  **if** $Rank == 0$ **then**
    Initialize model parameters $\theta_0$
  **end if**
  Broadcast $\theta_0$ from master to all processes
  *// Distributed Training (all processes execute)*
  Train model on *local_ data* to get $\theta_{final}$. The core synchronous gradient update mechanism is detailed in Algorithm 2.
  *// Distributed Evaluation (all processes execute)*
  Calculate local test error on *local_ data* using $\theta_{final}$
  Reduce all local errors to the master to compute global RMSE
  **if** $Rank == 0$ **then**
    Store the result (hyperparameters + global RMSE)
  **end if**
**end for**
**if** $Rank == 0$ **then**
  Analyze all results and identify the best model
**end if**

**Algorithm 2** Synchronous Distributed SGD using MPI

**Require:** Local Training Dataset $\mathcal{D}_{\text{train},i}$, Local Test Dataset $\mathcal{D}_{\text{test},i}$, Initial Parameters $\theta^{(0)}$, Learning Rate $\eta$, Total Processes $np$, Global Batch Size $M$.

Initialize MPI environment: Get Rank $i \in \{0, \ldots, np-1\}$ and size $np$.
**Initialize Model:** All processes initialize local parameter copy $\theta^{(0)}$.
**for** $k = 1$ to Max_Iterations **do**
  **for** each local mini-batch $B_{\text{local}} \subset \mathcal{D}_{\text{train},i}$, where $|B_{\text{local}}| = M/np$ **do**
    // 1. Local Gradient Computation
    Compute local gradient $\nabla\theta_{\text{local}}$ on $B_{\text{local}}$.
    // 2. Global Synchronization (Allreduce)
    **Aggregate Gradients:** Perform MPI_Allreduce($\nabla\theta_{\text{local}}, \text{SUM}$) to compute the sum of all local gradients $\nabla\theta_{\text{sum}}$.
    **Average Gradient:** Compute global average gradient $\nabla\theta_{\text{global}} = \frac{1}{np}\nabla\theta_{\text{sum}}$.
    // 3. Parameter Update
    **Update Parameters:** All processes update their local model: $\theta \leftarrow \theta - \eta \cdot \nabla\theta_{\text{global}}$.
  **end for**
  // 4. Global Evaluation and Early Stopping
  Compute local loss $L_{\text{local}}$ and $\text{RMSE}_{\text{local}}$ on $\mathcal{D}_{\text{test},i}$.
  MPI_Reduce local results to Rank 0 to compute global metrics.
  **if** Rank $i = 0$ and Early Stopping condition met **then**
    Broadcast Termination Signal.
  **end if**
**end for**

# 4 Results

## 4.1 Hyperparameter Tuning Results

### 4.1.1 Comprehensive Results of the Grid Search

To identify the optimal model configuration, a grid search was conducted across three activation functions (Sigmoid, Tanh, and ReLU), four batch sizes (16, 32, 128, and 256), and three hidden layer sizes (16, 32, and 64). In total, 45 combinations of hyperparameters were evaluated. The detailed results, verifiable by the execution logs in Figure 4, are summarized in Tables 3, 4, and 5, where each model was assessed based on its **Test RMSE** performance. The lower the RMSE value, the better the predictive accuracy of the model.

Table 3: Sigmoid Activation Function Grid Search Results

| Batch Size | Hidden Size | Test RMSE |
|---|---|---|
| 16 | 16 | 0.5438 |
| 16 | 32 | 0.5398 |
| 16 | 64 | 0.5404 |
| 32 | 16 | 0.5425 |
| 32 | 32 | 0.5397 |
| 32 | 64 | 0.5403 |
| 64 | 16 | 0.5419 |
| 64 | 32 | 0.5390 |
| 64 | 64 | 0.5397 |
| 128 | 16 | 0.5406 |
| 128 | 32 | **0.5389** |
| 128 | 64 | 0.5392 |
| 256 | 16 | 0.5439 |
| 256 | 32 | 0.5400 |
| 256 | 64 | 0.5391 |

*Best Configuration:* Batch Size = 128, Hidden Size = 32, Test RMSE = **0.5389**.

Table 4: Tanh Activation Function Grid Search Results

| Batch Size | Hidden Size | Test RMSE |
|---|---|---|
| 16 | 16 | 1.0635 |
| 16 | 32 | 0.7569 |
| 16 | 64 | 1.1762 |
| 32 | 16 | 1.0159 |
| 32 | 32 | 0.8194 |
| 32 | 64 | 0.8862 |
| 64 | 16 | 0.6549 |
| 64 | 32 | 1.3251 |
| 64 | 64 | 0.6573 |
| 128 | 16 | 0.5917 |
| 128 | 32 | 0.6214 |
| 128 | 64 | 0.6042 |
| 256 | 16 | 0.6195 |
| 256 | 32 | **0.5725** |
| 256 | 64 | 0.5731 |

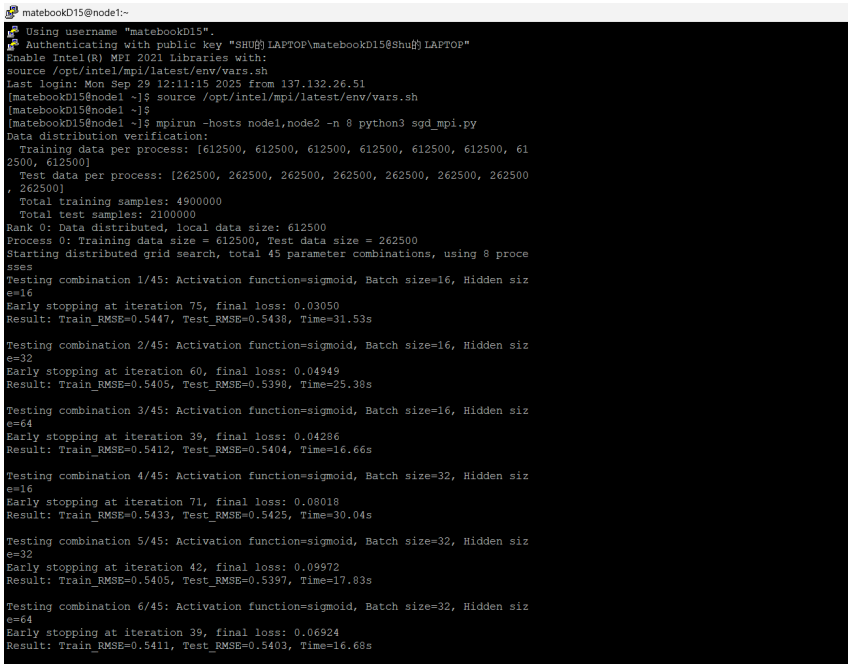
*Best Configuration:* Batch Size = 256, Hidden Size = 32, Test RMSE = **0.5725**.

Table 5: ReLU Activation Function Grid Search Results

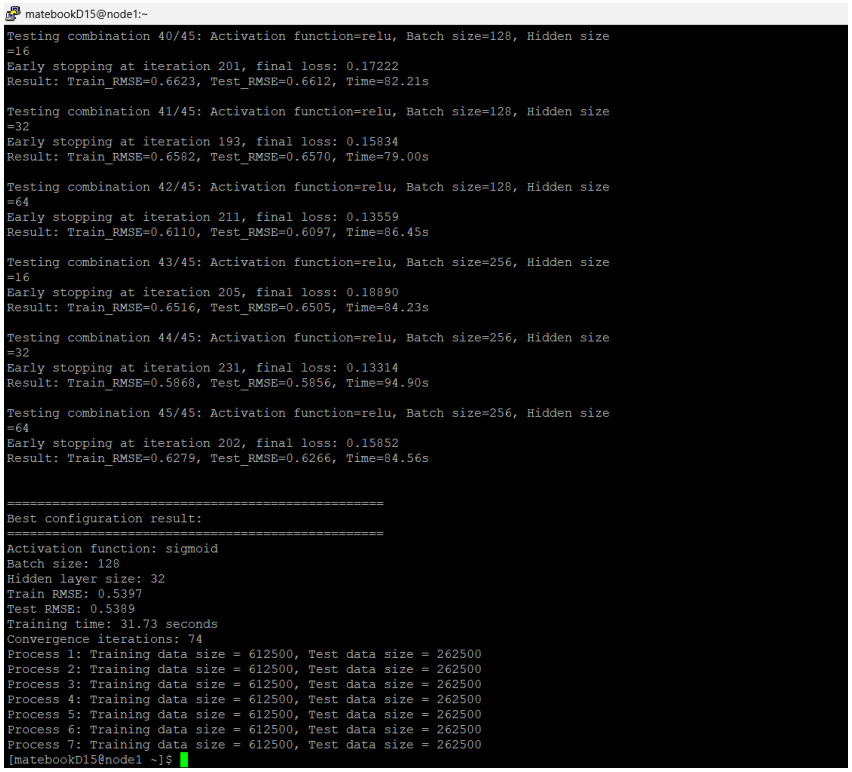
| **Batch Size** | **Hidden Size** | **Test RMSE** |
|---|---|---|
| 16 | 16 | 1.5159 |
| 16 | 32 | 1.3028 |
| 16 | 64 | 1.1588 |
| 32 | 16 | 0.7585 |
| 32 | 32 | 0.7276 |
| 32 | 64 | 0.7249 |
| 64 | 16 | 0.9691 |
| 64 | 32 | 0.6605 |
| 64 | 64 | 0.7798 |
| 128 | 16 | 0.6612 |
| 128 | 32 | 0.6570 |
| 128 | 64 | 0.6097 |
| 256 | 16 | 0.6505 |
| 256 | 32 | **0.5856** |
| 256 | 64 | 0.6266 |

*Best Configuration:* Batch Size = 256, Hidden Size = 32, Test RMSE = **0.5856**.

Figure 4: Distributed Grid Search Execution Logs (7 Million Sample Subset)




a) Top Section: Initial command invocation and results for the first few set of parameter combinations.




b) Bottom Section: Remaining results and summary of the final best configuration (Sigmoid, BS=128, HS=32) and its Test RMSE.

### 4.1.2 Discussion and Identification of the Optimal Configuration

According to the results summarized in Tables 3, 4, and 5, the **Sigmoid** activation function achieved the best overall performance in minimizing the Test RMSE. Among all tested config-

urations, the combination of a batch size of 128 and a hidden size of 32 produced the lowest RMSE value of 0.5389, suggesting that the Sigmoid function was particularly effective in capturing nonlinear relationships in the dataset under the given model architecture.

The **Tanh** activation function ranked second, with its best configuration (batch size 256, hidden size 32) achieving an RMSE of 0.5725. Although its performance fluctuated more across parameter combinations, it still demonstrated relatively strong predictive ability with larger batch sizes. In contrast, the **ReLU** activation function performed less effectively in this task, with its optimal configuration (batch size 256, hidden size 32) yielding an RMSE of 0.5856. This result may reflect that ReLU typically performs better in deeper or more complex neural network structures, while the present model was relatively shallow and thus could not fully exploit its advantages.

A consistent pattern observed across all three activation functions is that larger batch sizes (128 and 256) tended to produce lower RMSE values. This suggests that, in a distributed training environment or when handling large-scale data, larger batch sizes contribute to more stable gradient updates and improved generalization performance. Overall, the grid search indicates that the **Sigmoid activation function**, combined with **moderate hidden-layer complexity** and a **larger batch size**, offers the most balanced and accurate configuration for this experiment.

## 4.2 Final Performance and Training Process

### 4.2.1 Data Parallelism

The allocation of data forms the foundation of our data-parallel approach. The total dataset consists of 26,618,561 training samples and 11,407,955 test samples. Our program is designed such that the master process (Rank 0) can **automatically and approximately evenly partition and distribute the entire dataset to any number of MPI processes** ($np$). This is successfully demonstrated in our subsequent scaling tests with 8, 16, and 32 processes. Table 6 summarizes the specific distribution under the 8-process configuration, where each process handles roughly 3.32 million training samples and 1.42 million test samples, establishing the basis for our data-parallel implementation.

Table 6: Data Distribution per Process (8-Process Configuration)

| **Data Split** | **Sample Count** |
|---|---|
| Total Training Samples | 26,618,561 |
| Total Test Samples | 11,407,955 |
| Training data per process | 3,327,320 |
| Test data per process | 1,425,995 |

### 4.2.2 Final Model Performance

This section presents the validated performance metrics for the optimal hyperparameter configuration (Sigmoid, Batch Size=128, Hidden Size=32) identified through the preceding grid search on the data subset. For this final analysis, the model was trained using the **full dataset** on the 2-VM cluster with a total of **8 MPI processes**. These results provide the essential baseline for performance and stability, which will be used in Section 3.3 to assess the communication overhead and speedup gains across varying process counts.

The model training converged rapidly, triggering the early stopping criterion at the 49th iteration as the loss ceased to show significant improvement. The entire training process on the full dataset, which contains tens of millions of records, was completed in just 44.29 seconds(verifiable

in Figure 6). This efficiency highlights the significant advantage of distributed computing with 8 parallel processes for handling large-scale data.

The training loss curve for the final model is presented in Figure 5. The plot confirms a stable and effective training process, with the model's loss decreasing smoothly and stabilizing at a low value in under 50 iterations.

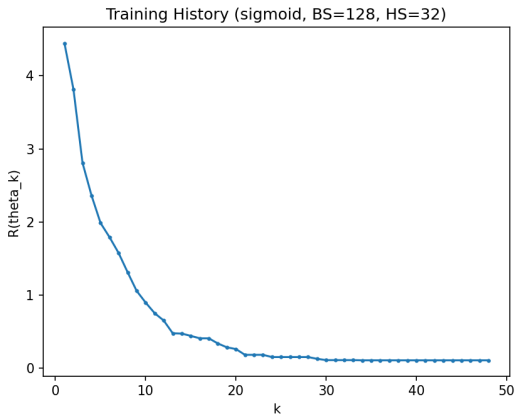



Figure 5: Training history of the final model on the full dataset (Sigmoid, BS=128, HS=32).

The primary metric for evaluating our model is the Test Root Mean Squared Error (RMSE), which achieved a value of 0.5398, indicating a high level of prediction accuracy. To assess generalization, we compared the training RMSE (0.5399) with the test RMSE (0.5398). The proximity of these two values is an ideal outcome, suggesting that the model did not overfit to the training data. The final model, configured with a sigmoid activation function, a hidden layer size of 32, and a batch size of 128, demonstrated strong generalizability and was trained in a remarkably short time. The key performance metrics are summarized in Table 7.

Table 7: Final Model Configuration and Performance Metrics

| **Parameter / Metric** | **Value** |
|---|---|
| Activation Function | Sigmoid |
| Batch Size | 128 |
| Hidden Layer Size | 32 |
| Training Time | 44.29 seconds |
| Convergence Iterations | 48 |
| Final Loss | 0.11042 |
| Train RMSE | 0.5399 |
| **Test RMSE** | **0.5398** |

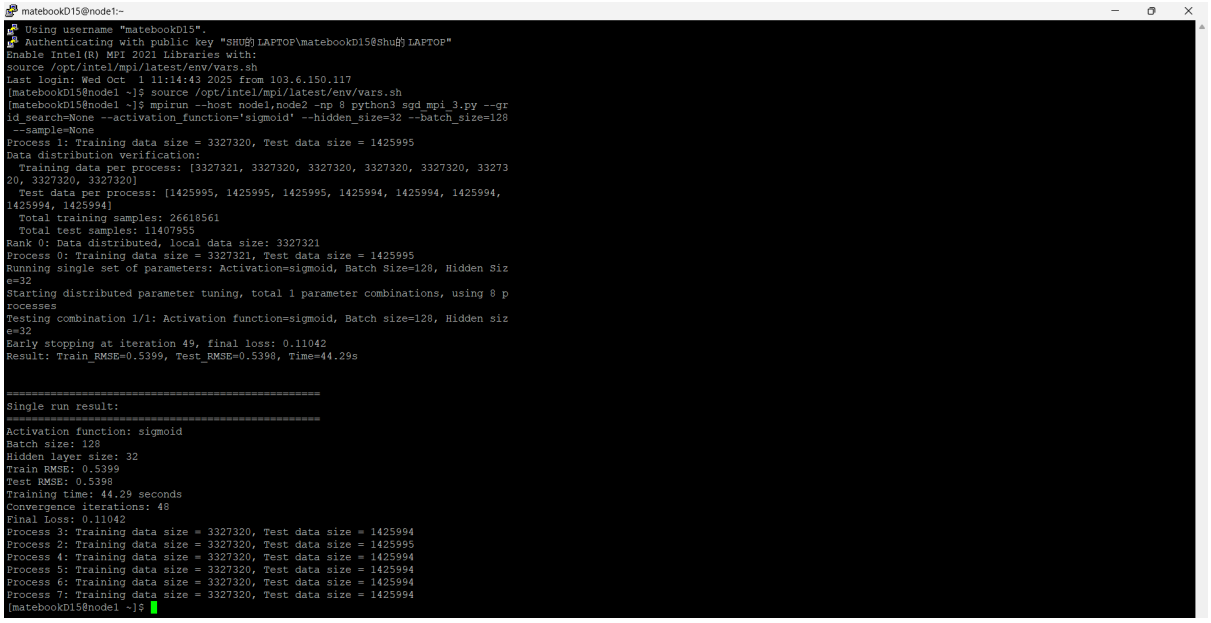



Figure 6: Command-line output showing the final performance run with 8 MPI processes. This verifies the total training time of **44.29 seconds** and convergence at the **48th iteration** for the optimal configuration (Sigmoid, BS=128, HS=32).

### 4.3 Analysis of the Impact of Process Count on Training Time

All scaling tests (8, 16 and 32 processes) were consistently performed using the **complete dataset** (26.6M training samples and 11.4M test samples). This ensures that the comparison of training time accurately reflects the efficiency gains of data parallelism on the full, production-scale workload.

To comprehensively evaluate the efficiency and scalability of the distributed training framework, we benchmarked the optimal hyperparameter configuration (Sigmoid, Batch Size=128, Hidden Size=32) across various numbers of MPI processes (8, 16, 32) on our dual-VM cluster.

The experimental results clearly demonstrated the acceleration achieved through MPI parallelization. For instance, the total training time for the model using **8 processes** was **44.29 seconds**. When the process count was increased to **16 processes**, the training time was significantly reduced to **28.83 seconds** (Convergence iterations: 46), representing an improvement of approximately 35%. Further scaling to **32 processes** reduced the training time to **26.92 seconds** (Convergence iterations: 42).

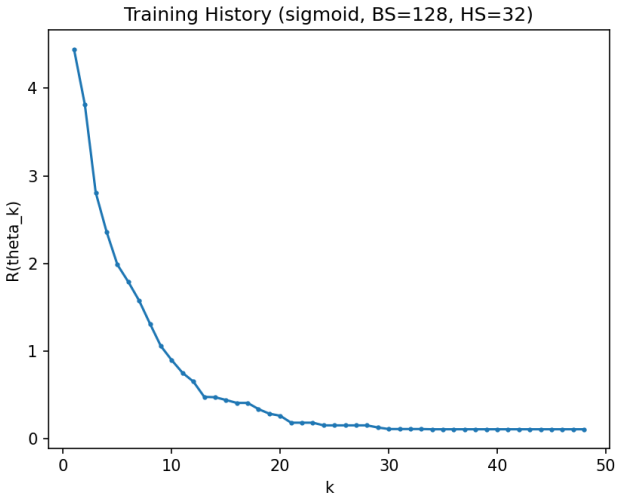
Table 8: Impact of MPI Process Count on Training Time (Optimal Hyperparameters)

| MPI Processes | Training Time (s) | Convergence Iterations | Speedup (vs. 8p) |
|---|---|---|---|
| 8 | 44.29 | 48 | 1.00× |
| 16 | 28.83 | 46 | 1.54× |
| 32 | 26.92 | 42 | 1.64× |

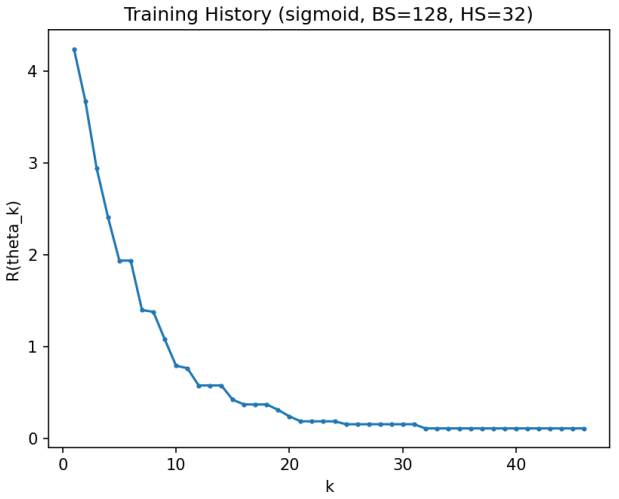
While the number of iterations required for convergence remained in a similar range across different process counts (48 iterations for 8 processes), the reduction in total training time was notable and consistent. This strongly validates that our MPI-based synchronous SGD implementation exhibits excellent **data parallelism scalability**, effectively utilizing cluster resources to handle large-scale datasets and significantly reduce the overall time required for model training.

To visually verify the stability and consistency of the model's convergence process across different process counts, we compared the training loss curves ($R(\theta_k)$ versus iteration count $k$) for the 8, 16, and 32 process configurations (see Figure 4). As the figure illustrates, despite the significant reduction in wall-clock training time, the shape of the loss curves across all configurations remained highly similar. They all smoothly decreased and stably converged to a very low loss value within a similar number of iterations. This further confirms that the synchronous SGD framework, when scaled up, successfully maintains the **stability and quality of training** while achieving **high-efficiency acceleration**.

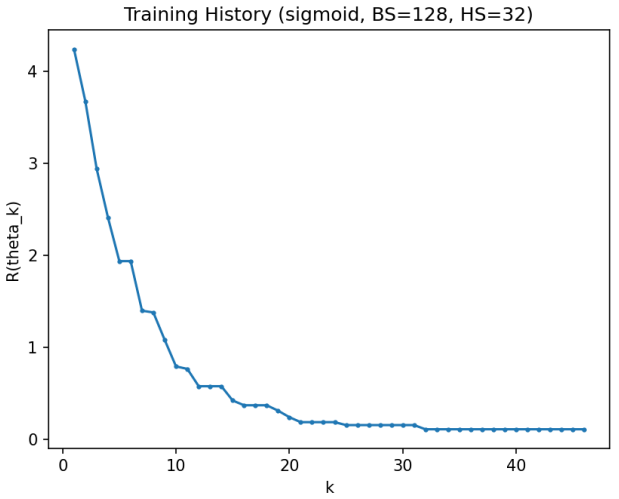
Figure 7: Comparative Training History for Different MPI Process Counts




a) 8 Processes




b) 16 Processes




c) 32 Processes

# 5 Discussion

## 5.1 Conclusion

In this project, we implemented a shallow neural network with embedding layers and trained it using stochastic gradient descent (SGD) across multiple processes with MPI. Our experiments demonstrated several key findings. First, embedding layers were helpful for handling high-cardinality categorical variables efficiently.

Second, refinement of the early stopping mechanism with a patience variable significantly stabilized the training, preventing premature termination and allowing the model to reach lower loss and better test RMSE.

Third, the distributed training with MPI yielded measurable benefits. By distributing full data and parallelizing both training and hyperparameter search, we were able to leverage multiple virtual machines and processes to address a large dataset and reduce overall runtime.

Overall, our results show that combining embedding-based feature representations, refined stopping mechanism, and distributed parallelism produces a scalable and effective pipeline for this large-scale regression problem. Future work could explore deeper architectures, adaptive learning rate strategies, or asynchronous gradient updates to further improve both model accuracy and distributed efficiency.

## 5.2 Future Work

The current synchronous SGD implementation relies on the robust MPI_Allreduce primitive for gradient aggregation. While highly optimized, an alternative communication pattern can be explored to potentially reduce computational redundancy and overhead, especially given the size of our model parameters (which include large embedding matrices).

**Reduce + Bcast Strategy for Parameter Synchronization** The future work will investigate replacing the integrated MPI_Allreduce operation with a two-phase, master-slave communication sequence for gradient synchronization:

1. **Reduce Phase:** All worker processes use MPI_Reduce to aggregate their locally computed gradients ($\nabla\theta$) onto the master process (Rank 0).
2. **Update Phase:** Rank 0 computes the global average gradient, executes the parameter update ($\theta_{k+1} = \theta_k - \eta\tilde{\nabla}R$), which includes the neural network weights and the embedding matrix.
3. **Broadcast Phase:** Rank 0 uses MPI_Bcast to distribute the **fully updated new model parameters** ($\theta_{k+1}$) back to all worker processes.

**Rationale:** This strategy primarily eliminates the **redundant parameter update computation** on worker processes. Specifically, by centralizing the parameter update on Rank 0, this approach optimizes the overall balance between communication and computation. Although MPI_Allreduce may offer superior communication latency, the Reduce + Bcast method provides a valuable trade-off by **significantly reducing the total amount of floating-point arithmetic** performed across the cluster in each iteration.

# 6 AI Communication

- **Question 1:** In a single-VM multi-process setup, how is the data distributed? Does each process really only own a subset of the data?

**Answer 1:** Through MPI's `Scatter` operation, the root process (Rank 0) splits the data into chunks and distributes them to each process. Each process only holds part of the data, and memory isolation is ensured between processes. Although all processes share the same physical memory on the VM, each process has its own virtual memory space. The operating system provides memory protection so that one process cannot directly access another process's memory.

- **Question 2:** For stochastic gradient descent with a mini-batch size of $M$, is $M$ sampled in total across all processes (i.e., each process takes $M/\text{np}$ samples), or does each process independently sample $M$ points?

  **Answer 2:** Two possible schemes exist:
  **(A)** A total of $M$ samples are drawn across all processes, so each process takes $M/\text{np}$ samples. Gradients from all processes then need to be aggregated to obtain the full gradient.
  **(B)** Each process independently samples $M$ points, equivalent to a global batch size of $M \times \text{np}$.
  The standard distributed SGD approach is scheme (A). Scheme (B) effectively enlarges the batch size, which can negatively affect convergence speed and generalization.

- **Question 3:** If we replace `allreduce` (where all processes perform the computation) with a design where only Rank 0 performs the final computation and other processes only handle data transfer, would this be better?

  **Answer 3:** This scheme can indeed be more efficient, because:
  1. **Centralized computation:** Only Rank 0 performs the final aggregation, avoiding redundant computations across processes.
  2. **Communication efficiency:** Using `reduce` is more efficient than `allreduce`, since results do not need to be broadcast back to all processes.
  3. **Resource utilization:** Other processes mainly handle communication, while computation resources are more efficiently allocated to Rank 0.

- **Question 4:** During preprocessing the full data, we encountered the memory error when we tried to transform all the categorical features into one-hot encoding, as it expands them into hundreds of columns. How can we address this problem?

  **Answer 4:** For categorical columns, we can treat them as categorical IDs, like words in NLP. They don't have a numeric meaning but one-hot encoding wastes huge memory. Instead, we can map each ID to a dense embedding vector. This is exactly like how we embed words in NLP and we then let the NN learn embeddings. That's much more memory-efficient.