DSA 5208 Project 1
Stochastic Gradient Descent for Neural Networks using MPI

Shreya Sriram, A0327236E

September 26, 2025

1 Introduction

1.1 Problem Statement

The problem statement is to use stochastic gradient descent to update the gradients of a one hidden layer neural network to minimize the training loss until it ceases to decrease any further.

For a given dataset,

$$\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{N}, \quad x^{(i)} = \left(x_1^{(i)}, \ldots, x_m^{(i)} \right) \in \mathbb{R}^m, \quad y^{(i)} \in \mathbb{R}$$

a neural network with one hidden layer approximates the map from x_i to y_i using the following equation:

$$f(x; \theta) = \sum_{j=1}^{n} w_j \sigma \left(\sum_{k=1}^{m} w_{jk} x_k + w_{j,m+1} \right) + w_{n+1}$$

where

$$\begin{aligned} \theta = (&w_1, \ldots, w_n, w_{n+1} \\ &w_{11}, \ldots, w_{1,m+1} \\ &\ldots \\ &w_{n1}, \ldots, w_{n,m+1}) \end{aligned}$$

is the set of all parameters in the model and $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear activation function. The stochastic gradient method aims to solve the following minimization problem:

$$\min_{\theta} R(\theta) = \frac{1}{2N} \sum_{i=1}^{N} \left| f\left(x^{(i)}; \theta \right) - y^{(i)} \right|^2$$

To find the optimal value of θ, we start with an initial guess θ_0, and update the solution with

$$\theta_{k+1} = \theta_k - \eta \widetilde{\nabla_\theta R} (\theta_k) , \tag{1}$$

where η is the learning rate, and $\widetilde{\nabla_\theta R}$ is the approximation of the gradient

$$\nabla_\theta R(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[f\left(x^{(i)}; \theta \right) - y^{(i)} \right] \nabla_\theta f \left(x^{(i)}; \theta \right) .$$

The approximation is done by randomly drawing M distinct integers $\{j_1, \ldots, j_M\}$ from the set $\{1, \cdots, N\}$ and setting $\widetilde{\nabla_\theta R}$ to

$$\widetilde{\nabla_\theta R}(\theta) = \frac{1}{M} \sum_{i=1}^{M} \left[f\left(x^{(j_i)}; \theta \right) - y^{(j_i)} \right] \nabla_\theta f \left(x^{(j_i)}; \theta \right)$$

The random set $\{j_1, \ldots, j_M\}$ must be updated for every iteration (1). The iteration terminates when $R(\theta_k)$ no longer decreases.

1.2 Dataset

The dataset we want to execute the solution to the problem statement is ny-taxi2022.csv which has 39656098 rows and 19 attributes. The attributes of which are described as below (source: Kaggle) :

1. VendorID: A unique identifier for the taxi vendor or service provider.
2. `tpep_pickup_datetime`:
   The date and time when the passenger was picked up.
3. `ttpep_dropoff_datetime`: The date and time when the passenger was dropped off.
4. `tpassenger_count`: The number of passengers in the taxi.
5. `ttrip_distance`: The total distance of the trip in miles or kilometers.
6. `tRatecodeID`: The rate code assigned to the trip, representing fare types.
7. `store_and_fwd_flag`: Indicates whether the trip data was stored locally and then forwarded later (Y/N).
8. `PULocationID`: The unique identifier for the pickup location (zone or area).
9. `DOLocationID`: The unique identifier for the drop-off location (zone or area).
10. `payment_type`: The method of payment used by the passenger (e.g., cash, card).

11. `fare_amount`: The base fare for the trip.
12. `extra`: Additional charges applied during the trip (e.g., night surcharge).
13. `mta_tax`: The tax imposed by the Metropolitan Transportation Authority.
14. `tip_amount`: The tip given to the driver, if applicable.
15. `tolls_amount`: The total amount of tolls charged during the trip.
16. `improvement_surcharge`: A surcharge imposed for the improvement of services.
17. `total_amount`: The total fare amount, including all charges and surcharges.
18. `congestion_surcharge`: An additional charge for trips taken during high traffic congestion times.

1.3 Requirements

- The code should work for any number of processes.
- The dataset is stored nearly evenly among processes, and the algorithm should not send the local dataset to other processes, except when reading the data.
- Split the dataset into a training set (70%) and a test set (30%).
- Predict the total fare amount (column total_amount) using the following columns as features:
  `tpep_pickup_datetime`, `tpep_dropoff_datetime`, `passenger_count`, `trip_distance`, `RatecodeID`, `PULocationID  DOLocationID`, `payment\_type`, `extra`
- You may need to preprocess of the data by dropping some incomplete rows and normalizing the data.
- All processes should compute the stochastic gradient $\widetilde{\nabla_\theta R}$ in parallel.
- Once the solution θ is found, the code can compute the RMSE in parallel.

2 Main Approach

2.1 Exploratory Data Analysis

Since the given dataset was huge to effectively process locally on my machine in a distributed setting, I performed the following exploratory data analysis (not distributed) in a Jupyter notebook:

1. I loaded the input file using the pandas' library's `read_csv` function while parsing the datetime attributes `tpep_pickup_datetime`, `tpep_dropoff_datetime` the first time and cached the entire dataframe as a pickle file to speed up subsequent runs

2. I checked the percentage of missing data in columns across the dataframe which was 3.450423 %, since the fraction was small, I chose to drop the rows with any missing data

3. I checked for duplicate rows to prevent bias during training. Since there was only 1 duplicate row, I dropped that row

4. I converted the pickup/dropoff datetime attributes to year, month, date, hour, minute, second and computed the trip duration in minutes between the pickup and dropoff times. I subsequently dropped the original datetime attributes as the derived attributes are sufficient to use for training and evaluation purposes.

5. I dropped the rows where `trip_duration` was either non-positive or over 3 hours to eliminate any noisy/invalid data

6. I dropped the rows with negative values of `extra` and the non-positive values of `total_amount`. As I further analyzed the range of values of `total_amount`, I noticed extremely high values and hence chose to winsorize 0.5 % of the dataset i.e. clipped the data between 0.5 percentile and 99.5 percentile to be conservative with the removal of the outliers. This cleared up 0.9394557120998955 % of the dataset.

7. I checked the frequency counts of the categorical variables i.e. `RatecodeID`, `payment_type`, `PULocationID`, `DOLocationID` to determine the encoding to use for them. Since `RatecodeID`, `payment_type` only have 7 and 6 values respectively, I chose to perform one-hot encoding for them. Since `PULocationID`, `DOLocationID` have 262 values each, one hot encoding could significantly impact training times hence I chose frequency encoding for them to keep the dimensionality low.

8. I performed a final round of checks to ensure the data was clean before writing the data to another csv for modeling the Neural Net.

The exploratory data analysis concluded with a reduction from 39656098 rows to 37603658 rows through removal of outliers, duplicates, invalid data alongside the necessary transformation of given columns through datetime attribute extraction, encoding etc. to have 30 features (excluding the response variable `total_amount`)

2.2 Distributed System

The distributed system comprises the following key components that each execute in parallel across different processes:

- IO and data splitting between train and test : Reading the processed input from the previous step and splitting the data between test and train
- Normalizer: Normalizing the train features and labels, using their mean and stddev to normalize the test data
- Neural Network : Trains the model using Stochastic Gradient Descent for batch updates on the test data until the loss is within a threshold and evaluates the model using the test data.

2.2.1 IO and splitting the data between test and train

The IO and test-train split can be summarized as below:

Since one of the requirements is to store data nearly evenly among processes, the process with rank 0 counts the number of rows in the input file and then broadcasts the count to the other processes using the MPI broadcast function:

$\texttt{num\_rows\_total} = comm.bcast(\texttt{n\_rows}, root = 0)$

Since the other processes are aware of the total number of rows, each process locally determines the number of rows it needs to read based on its rank. The start index, end index and skiprows are computed using the rank, and this makes the split among all processes nearly uniform.

To make the reading more efficient, we perform a chunk-based reading of the processed file with a chunk size of 100000

For each process, their local chunks are split into test and train - this is done by generating a permutation of indices using RNG and splitting the indices between test and train.

2.2.2 Normalizer

Within each process, the normalizer executes the standard scaling logic, i.e. centering the data to the global training mean and dividing by the global training stddev (non-zero). The global training mean and global training stddev for the training features are computed using the MPI allreduce function:

`feature_fields` $= comm.allreduce($(`local_feature_sum, local_feature_count`)$, op = MPI.SUM)$
`global_feature_sum`, `global_feature_count` = `feature_fields[0]`, `feature_fields[1]`

The global training mean and global training stddev for the training labels are computed in a similar manner.

These means and stddevs are used to normalize the testing data.

Since I was looking to determine the impact of not normalizing the feature-encoded attributes, I added the functionality of skipping normalization for certain columns to evaluate the impact of doing so on the experiments.

2.2.3 Neural Network

The One Hidden Layer Neural Network has the following steps that are applied in the training stage:

1. Forward propagation: Starting off with random weights, the input data (i.e. features from the processed data) is passed on to a hidden layer which then generates an output. So the activation is calculated on the weighted sum of inputs which on another weighted summation forms the output

2. Backward propagation: The predicted output generated by forward propagation is then compared against the actual output to adjust the weights and the bias at the hidden layer and then backpropagate it to adjust the weights and bias at the input layer using derivative of the chosen activation function.

3. Gradient update: The weights are adjusted based on the applicable learning rate and gradients. The gradients are computed at a batch level (using M integers drawn at random), they are the calculated using `global_summed_gradients` and `global_count`, obtained by MPI's allreduce function on local weighted gradients (weighted as the sizes of batches may vary) and local count respectively.

These steps are continuously repeated within a batch until the difference between the loss computed for a batch of data and its previous batch are within a certain chosen threshold ($1e-6$). This approximation of global loss at the batch level is used as a stopping criterion to exit training. The train RMSE is calculated using the updated weights/bias on the full train dataset.

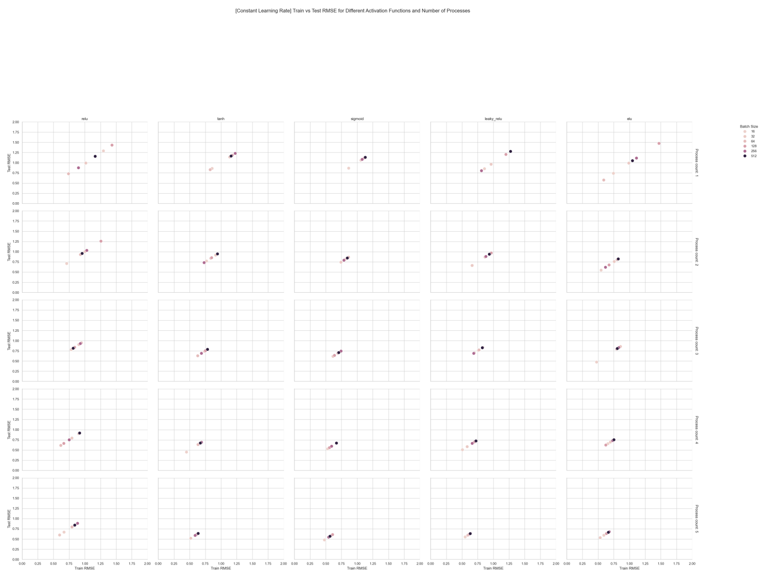
The evaluation stage uses the weights computed in the training stage to predict the labels, the test RMSE is computed similar to how the train RMSE is computed.

Additional notes:

1. Weight initialization: The starting weights are initialized based on the activation function to ensure faster convergence following (Reference: DeepLearning.AI Weight Initialization)
2. Learning rate initialization: I tried three types of learning rates - a constant learning rate, a cyclic scheduler learning rate (Reference: Learning Rate Schedulers) and a modified cyclic learning rate (that uses number of processes as a factor), details of the results are in the next section.

3 Results

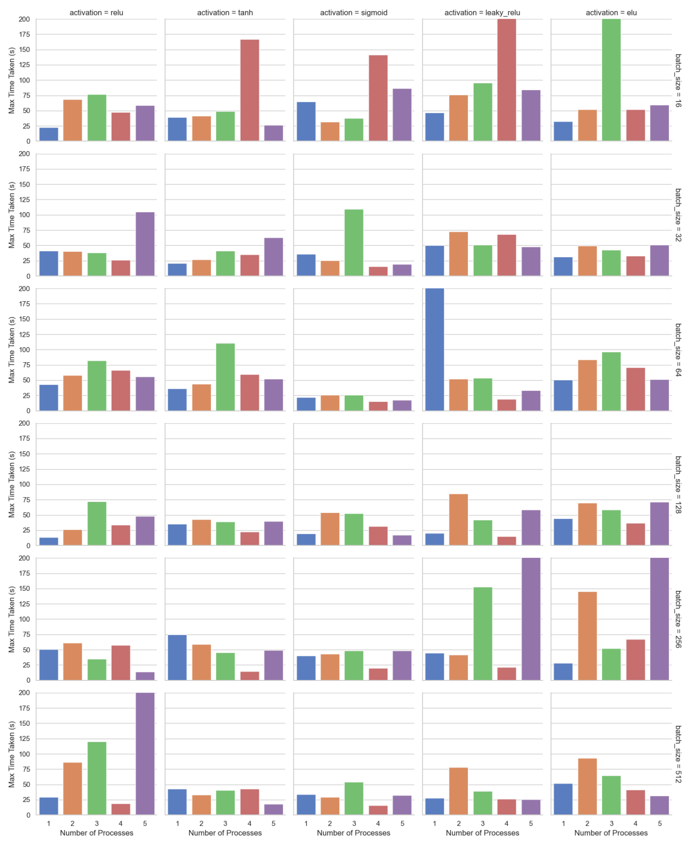
The results from the One Hidden Layer Neural Network model using the constant learning rate indicate that the training model generalizes well on the test data as training RMSE and testing RMSE are nearly equal across the board. The plot below is a scatterplot between train RMSE and test RMSE for the 5 activation functions I tried (`relu, tanh, sigmoid,` `leaky_relu, elu`) and by the number of processes (1 through 5) - the datapoints in each of the subplots correspond to the batch sizes as indicated in the legend of the grid plot.

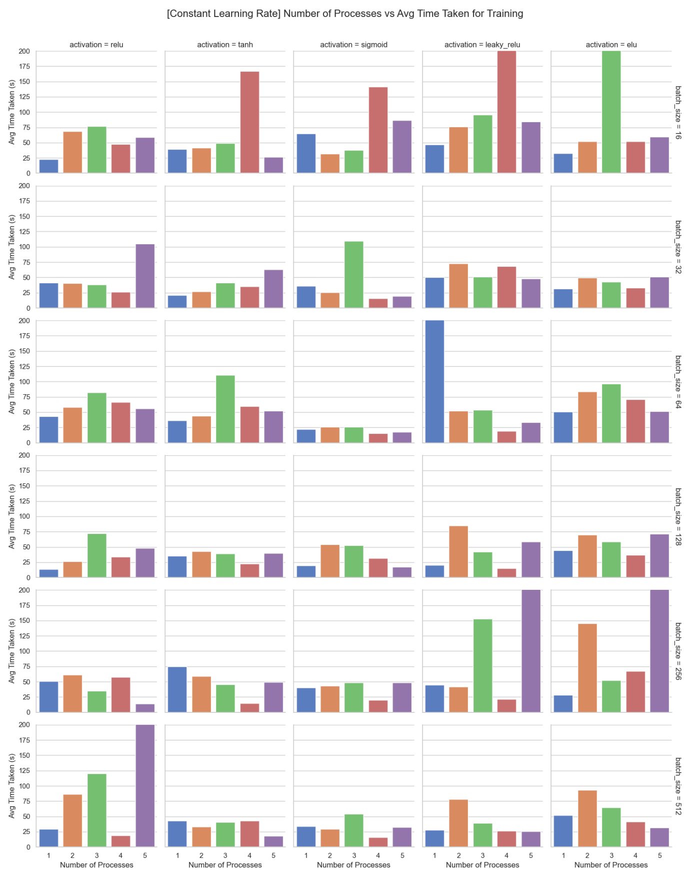


However, the constant learning rate results in early convergence resulting in high RMSEs.

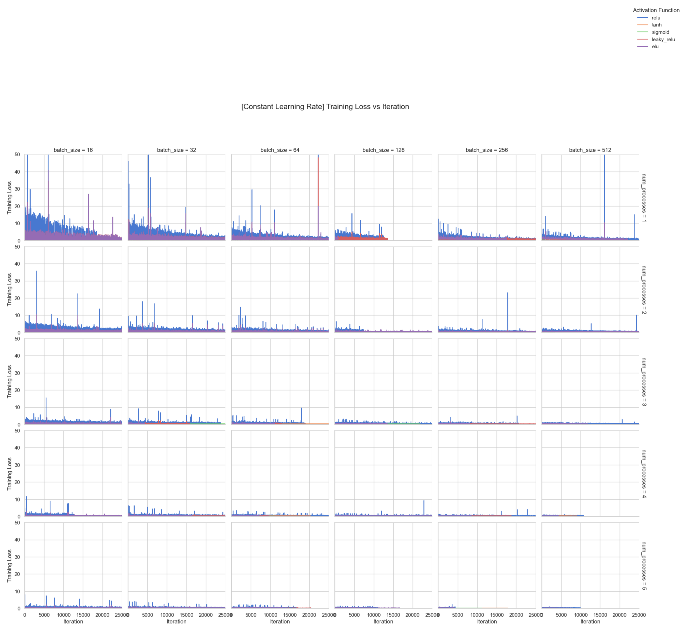
The following two plots are i) a bar chart between number of processes and the corresponding max training time (max training time is to measure the time taken by the slowest process) and ii) a bar chart between number of processes and the corresponding avg training time. These plots indicate that parallelism works reasonably well, although there are some spikes in times likely due to my local machine's resource (CPU/memory) limitations being hit.

[Constant Learning Rate] Number of Processes vs Max Time Taken for Training

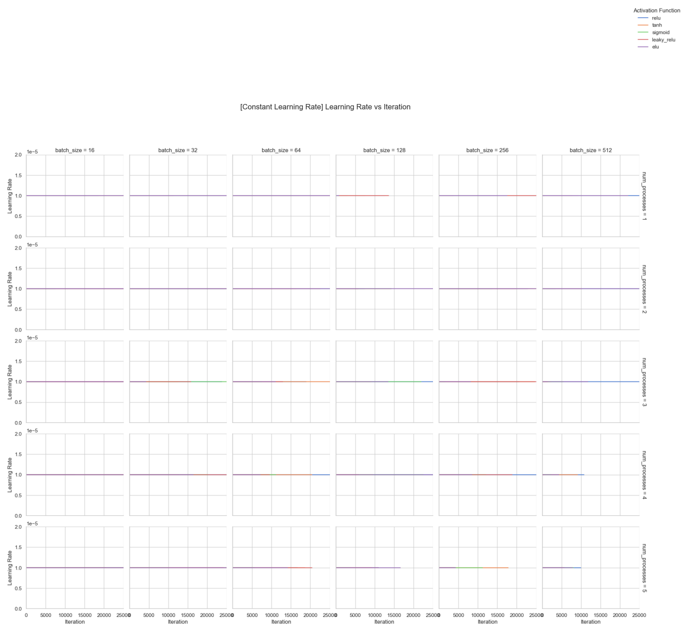




The following plot is a line chart between training loss and number of iterations, except for a few spikes when the process count is 1, the training loss is fairly stable with increasing number of iterations.



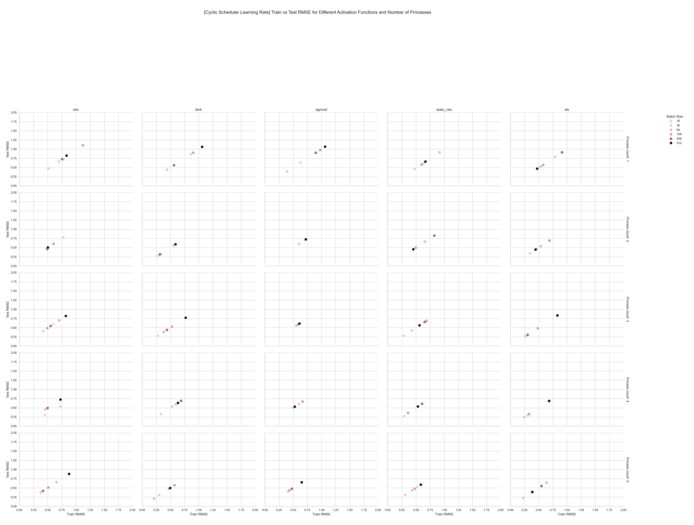
The learning rate is constant as the name suggests, and the same is indicated by the plot below.



4 Attempts to improve RMSE and convergence

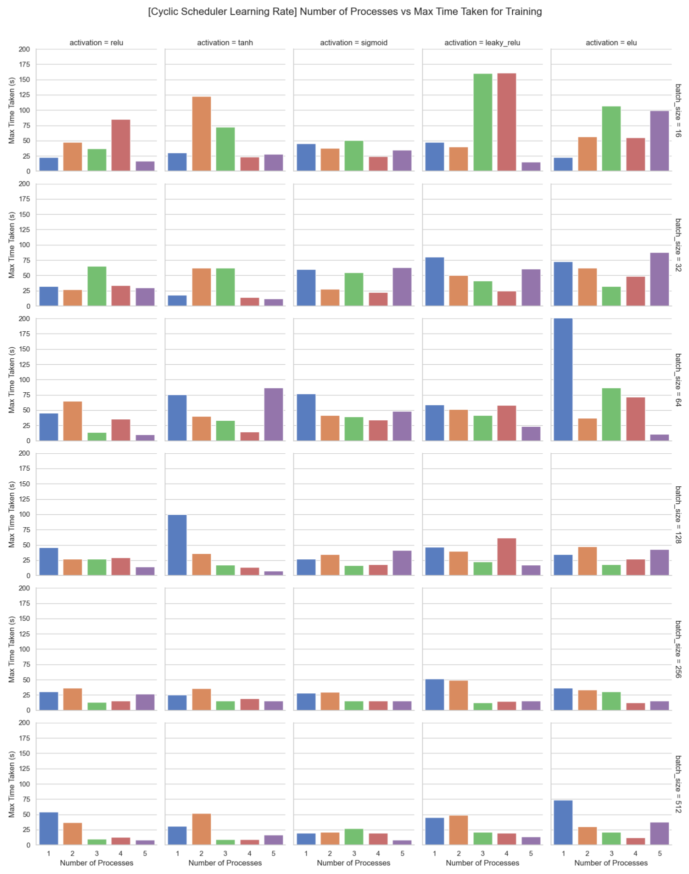
4.1 Attempt 1: Cyclic Scheduler Learning Rate

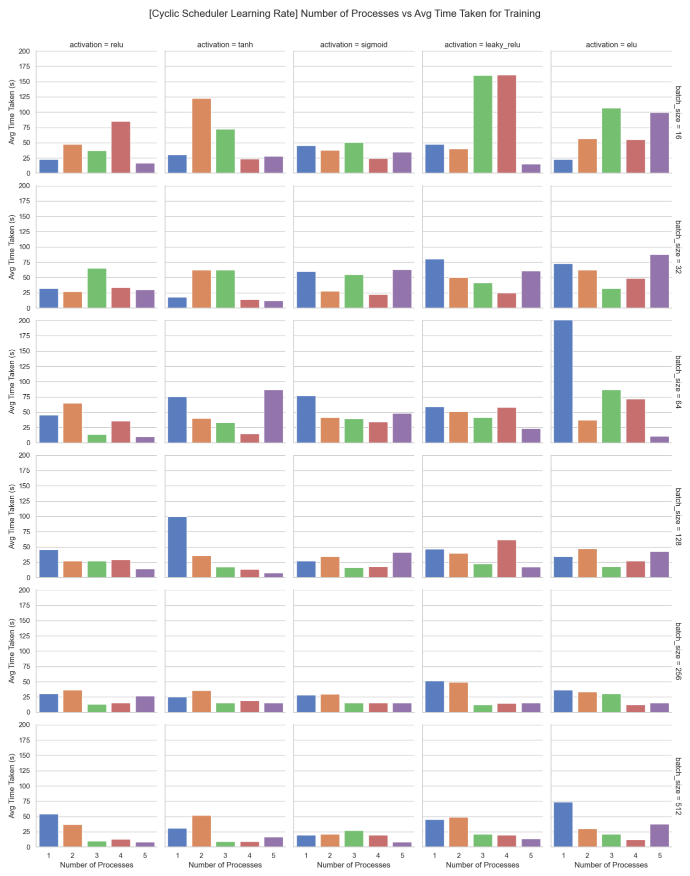
In an attempt to slow down convergence and have lower RMSE values, I tried using cyclic scheduler with a base learning rate of 1e-5, step size of 500 and max learning rate of 2e-4 i.e. the learning rate would keep cycling between the base learning rate and max learning rate enabling gradient adjustment in a suitable way over multiple (cycles of) iterations.



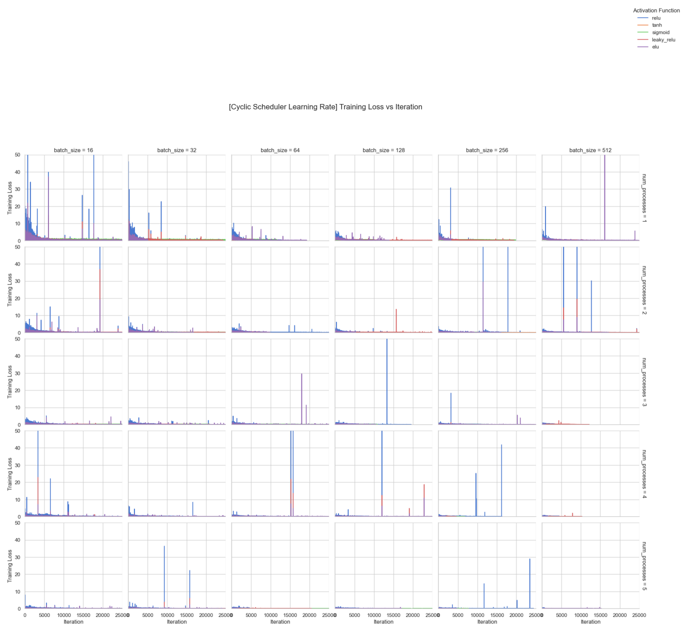
There is a visible improvement in the RMSE values compared to the constant learning rate.

The following two plots are bar plots for (i) max time taken for training vs number of processes, ii) avg time taken for training vs number of processes. While there are signs of parallelism working (relu for batch size =128 / 512, sigmoid/`leaky_relu` for batch size=256), there are multiple instances where time taken isn't monotonically decreasing with the number of processes thereby suggesting that we are unable to maximize the parallelization capabilities to reduce the time taken with the chosen learning rate function.

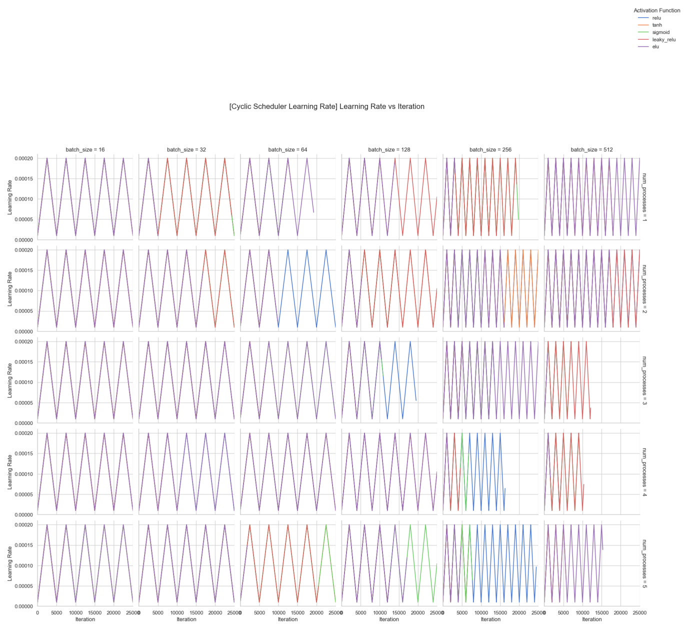

[Cyclic Scheduler Learning Rate] Number of Processes vs Max Time Taken for Training
activation = relu
activation = tanh
activation = sigmoid
activation = leaky_relu
activation = elu
batch_size = 16
batch_size = 32
batch_size = 64
batch_size = 128
batch_size = 256
batch_size = 512
Max Time Taken (s)
Number of Processes



Compared to the line chart for constant learning rate, the suggesting that the cyclical scheduler is enabling the model to quickly switch back from large spikes that could mean higher loss thanks to the cyclic nature of the learning rate. We also notice a slower convergence rate in certain permutations of parameters `(activation_type, num_processes)` - `((tanh,2 ), (leaky_relu, 2), (elu, 3), (elu, 4), (elu, 5))`

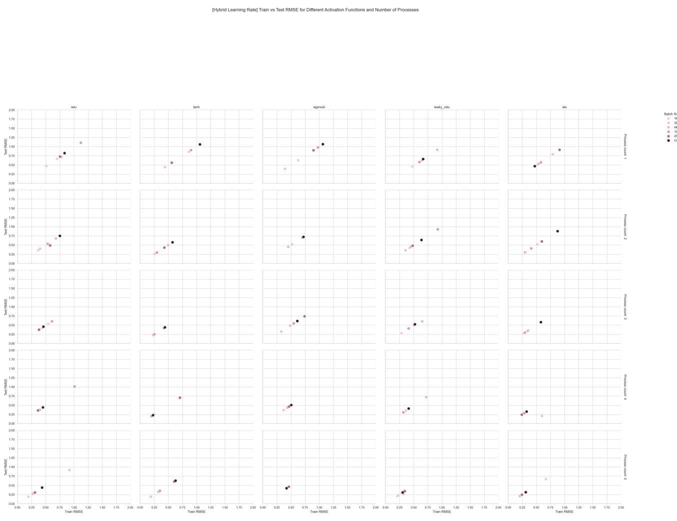


The learning rate is cyclic as the name suggests, with an adjustment by batch size and the same is indicated by the plot below.



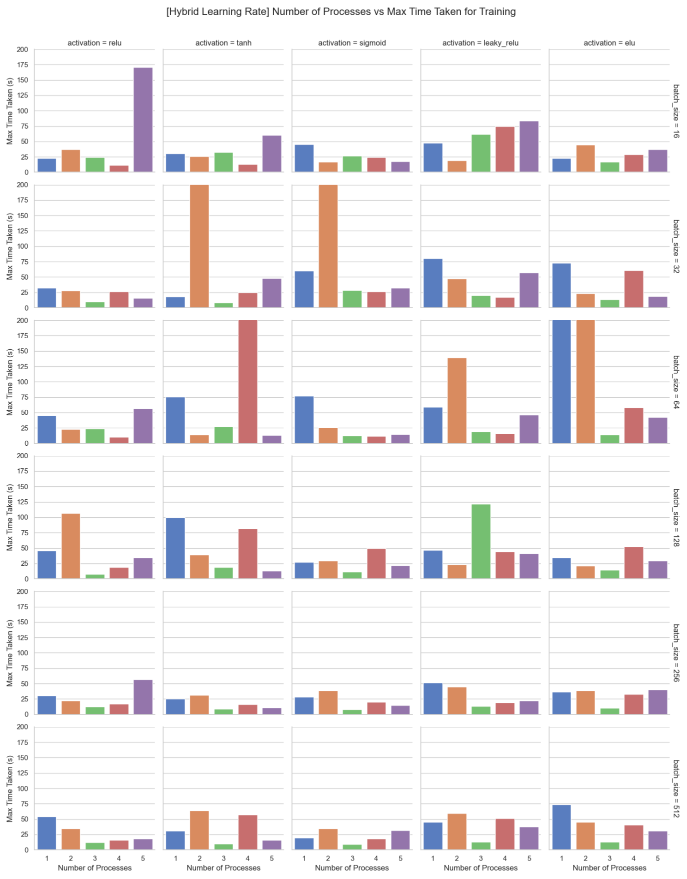
4.2 Attempt 2: Custom Learning Rate

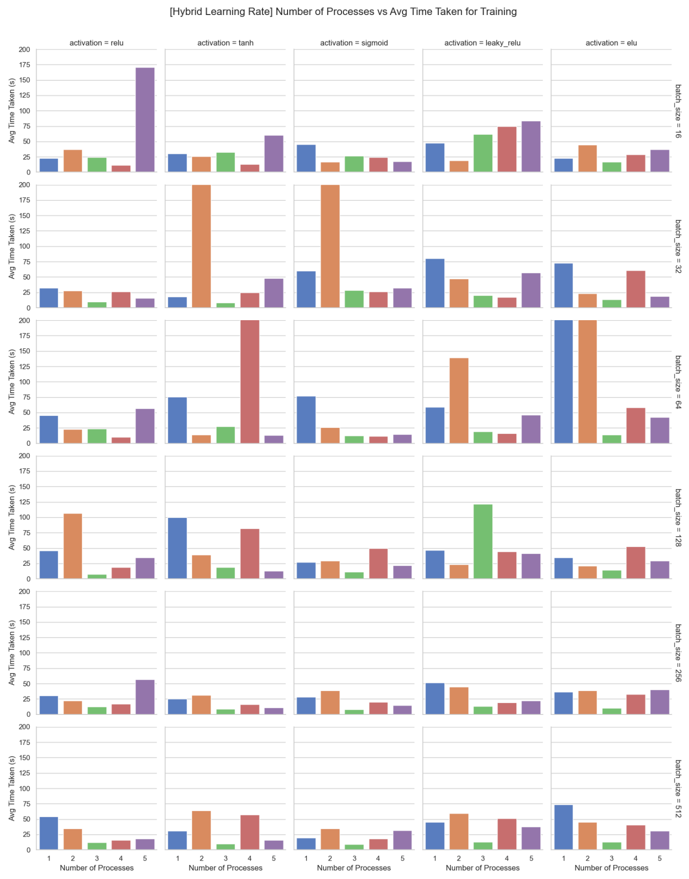
This attempt was to purely experiment with the learning rate to see if a custom function using a cyclic function with an additional factor of the number of processes could assist with having even lower RMSE values while also enabling more effective use of parallel processes; I modified the cyclic scheduler from the previous attempt to include a factor of size (size being the number of processes in the MPI notation).



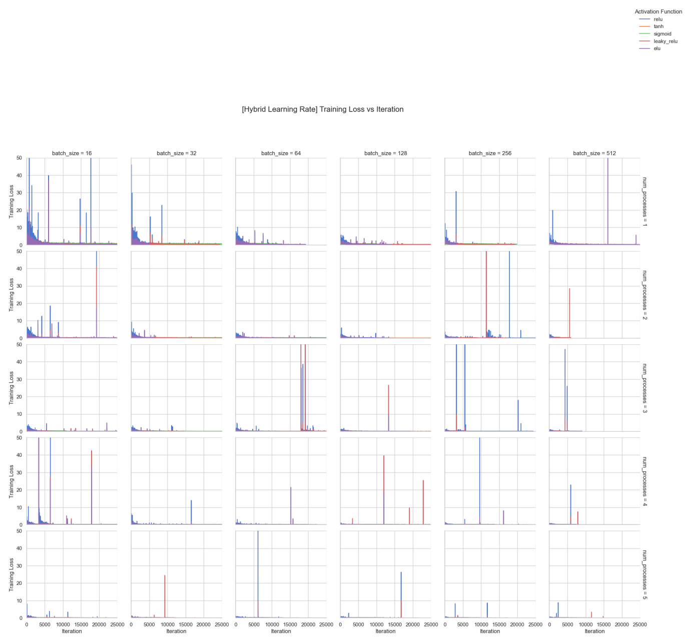
There is further improvement in the RMSE values compared to the previous two attempts specifically when there are multiple processes at play - `leaky_relu, elu` have some of the lowest RMSEs when we train/test the model across batches of all different sizes.

The following two plots are bar plots for (i) max time taken for training vs number of processes, ii) avg time taken for training vs number of processes. While there are signs of parallelism working (relu for batch size =128 / 512, sigmoid/`leaky_relu` for batch size=256), there are multiple instances where time taken isn't monotonically decreasing with the number of processes thereby suggesting that we are unable to maximize the parallelization capabilities with the current learning rate. The custom learning rate still suffers from arbitrary spiky max/avg times owing to the learning rate having non-uniform jumps.

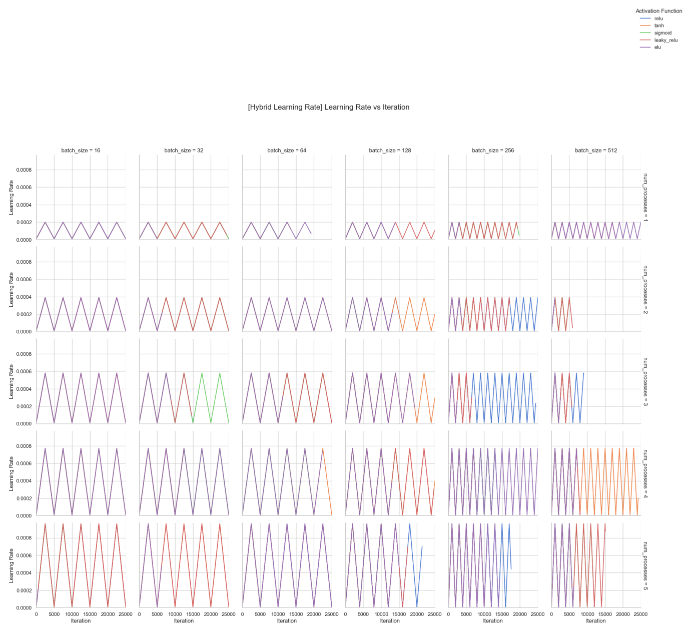

[Hybrid Learning Rate] Number of Processes vs Max Time Taken for Training
activation = relu
activation = tanh
activation = sigmoid
activation = leaky_relu
activation = elu
batch_size = 16
batch_size = 32
batch_size = 64
batch_size = 128
batch_size = 256
batch_size = 512
Max Time Taken (s)
Number of Processes



Compared to the line chart for cyclic learning rate, barring two combinations (tanh, 1), (elu, 1), the learning rate is .



The learning rate is still essentially cyclic as the name suggests, and it fluctuates with both the batch size and the number of processes is indicated in the plot below.



5 Future Work

Future work would involve the following:

(1) Further exploration of learning rates that could tap on parallel processing effectively while further lowering the RMSE values to ensure more accurate predictions on unknown data,
(2) Since the testing process was limited to a certain batch sizes and number of processes, running my code on a virtual machine on a hosted platform would allow for testing with higher batch sizes/number of processes thereby enabling effective testing of parallelism.