THE LONDON SCHOOL OF ECONOMICS AND POLITICAL SCIENCE

Department of Statistics

Group Project Report

Exploration of the costs and benefits of Parallelisation to Stochastic Gradient Descent

Candidate Number **42066**
Candidate Number **39180**
Candidate Number **43729**
Candidate Number **51348**
Candidate Number **45564**

2024/25

ST444 Computational Data Science

Repository Link: `https://github.com/shreyashonnalli/ST444`

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Optimization is at the heart of Machine Learning, where the objective is to minimize a loss function that quantifies the discrepancy between model predictions and real observations. **Gradient Descent** is one of the most fundamental and popular algorithms in this space. It is used in a broad range of Machine Learning (ML) tasks such as Linear and Logistic Regression, Support Vector Machines, and Neural Network architectures. Assuming differentiability of the loss/objective function, Gradient Descent utilizes the first-order derivatives of such a function w.r.t. the parameters to iteratively update said parameters by moving in the direction of steepest descent.

There are many variants of gradient descent that arise from ML research[1]. We will discuss the 2 fundamental types of the Gradient Descent algorithms:

1.1.1 Batch/Vanilla Gradient Descent

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta J(\theta; \mathbf{X}, \mathbf{y})$$

Batch Gradient Descent computes the gradient of the objective function $J(\theta)$ w.r.t the parameters θ for the whole dataset, and moves in the direction of steepest descent with step size η. In the modern era, the volume of available data has grown exponentially, with some industrial datasets reaching petabyte scales. Using the entire dataset to update model parameters is often infeasible due to memory constraints and computational overhead. Even when feasible, such updates become prohibitively expensive for large datasets, as each iteration requires significant processing time. Consequently, this approach is neither practical nor efficient for modern machine learning tasks. However, this method has the guarantee of converging to a global or local minimum depending on the convexity of the loss function.

1.1.2 Stochastic Gradient Descent (SGD)

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta J(\theta; x_i, y_i)$$

SGD on the other hand performs a parameter update for a randomly sampled instance and corresponding label from the training set in each iteration. Each update using SGD is much faster than vanilla gradient descent, assuming a large dataset. One property of SGD is significant fluctuations in the gradient estimates calculated at each iteration (high variance). This is caused by the fact that each data point used to update the parameters is randomly sampled. It also has the property to enable the parameters to "jump" out of local minima to lower points along the objective function. This property complicates the convergence to an exact minimum, but it has been shown that SGD has the same convergence behaviour as vanilla gradient descent as the step size slowly decreases [1].

Please note that when referring to SGD, most libraries typically refer to mini-batch gradient descent, where we use a subset of the dataset to iteratively update model parameters. This method is more popular than

either of the 2 gradient descent algorithms above as it takes advantages from both methods for large dimension/dataset problems.

1.1.3 Potential for Parallelisation

Although gradient descent is generally a sequential algorithm, parallelism is conceived to speed up the convergence rate of gradient descent, especially SGD. In the scenario where the data set is extremely large, it becomes computationally infeasible to process all the data via SGD on a single machine if we want to obtain a reasonable model in a matter of hours instead of days or weeks[2]. It is proposed by Zinkevich, Weimer, Li, *et al.* that parallelism could reduce the variance of the estimates and result in a more accurate approximation in the same wall clock time.

Numerous previous attempts have been made to optimize gradient descent through parallelism. We will now discuss the algorithms used in these papers and the results achieved, before ultimately moving on and designing our own experiments.

1.2 Background Research

In the previous section, we introduced SGD and why parallelism may benefit it. Here we introduce previous attempts at parallelising SGD, and briefly state their findings.

Zinkevich, Weimer, Li, *et al.* proposed an algorithm that was remarkably simple, and had many favourable theoretical properties such as "worst-case bounds which are similar SGD in terms of wall clock time, but vastly faster in terms of overall time" [2]. Furthermore, when they applied it to a large-scale dataset, they were able to empirically show a reduction in wall clock time to obtain a specific root mean square error (RMSE):

Algorithm 1 SGD($\{c^1, ..., c^m\}, T, \eta, \omega_0$)

for $t = 1$ to T **do**
  Draw $j \in \{1, ..., m\}$ uniformly at random
  $\omega_t \leftarrow \omega_{t-1} - \eta \partial_\omega c^j(\omega_{t-1})$
end for
return ω_T

Algorithm 2 ParallelSGD($\{c^1, ..., c^m\}, T, \eta, \omega_0, k$)

for $i \in \{1, ..., k\}$ **in parallel do**
  v_i = SGD($\{c^1, ..., c^m\}, T, \eta, \omega_0$) on client machine
end for
Aggregate from all computers $v = \frac{1}{k} \sum_{i=1}^{k} v_i$
return v

Algorithm 1 is implemented in parallel across multiple computers, which explains why increasing the number of machines does not result in an increase in wall clock time. In this work, we investigate the empirical behaviour of the algorithm in the context of linear regression. In contrast to the paper, we parallelize within a single machine using multiple processes. We hypothesize that wall clock time will increase as the number of processes grows, even with multiple cores available, due to the additional overhead introduced by the system scheduler, which must manage finite CPU resources and handle context switching between processes. We refer to this approach as **Parallelism through Processes**.

The Stochastic Gradient Descent with K samples algorithm given below was taken from the paper "Parallelizing Gradient Descent" (by Santos and Ojha), a project done for the course - Parallel Computer Architecture and Programming, Fall 2018 conducted at CMU, School of Computer Science, ([3]). In this work, we conducted an experiment inspired by our understanding of this algorithm. We understood this algorithm to be equivalent to parallelizing mini-batch gradient descent which is often seen as a balance between stochastic gradient descent and batch gradient descent. In mini-batch gradient descent, we consider a batch of fixed number of data points (smaller than the whole dataset), a "mini-batch" and computes the gradient of the objective function $J(\theta)$ w.r.t the parameters θ for the mini-batch, and moves in the direction of steepest descent. Santos and Ojha observed faster convergence rates with a small number of additional processors but found that the performance reached a limit with a higher number of processors (after 8 or 16 threads).

Algorithm 3 SGD with k samples

Randomly initialise parameters
for j **do** in range(iterations)
    Sample $(k * no\_of\_threads)$ points from original dataset
    Split sample into batches of k
    Compute gradient for each batch in parallel using $(no\_of\_threads)$ threads
    Compute average gradient across batches
    Update parameter
end for

One of the fundamental assumptions of Stochastic Gradient Descent (SGD) is that points from the input data (X) are selected independently and randomly. However, this random selection process can be computationally expensive and may also lead to cache misses when working with very large datasets. A solution proposed by Meng, Chen, Wang, *et al.* [4]. is outlined bellow.

Algorithm 4 SGD with Shuffling

Input: Data of size n, number of threads t, number of epochs i
Initialize: Randomly initialize the parameter β_0
Randomly shuffle the input data
Split the shuffled data into t groups, $G_1, G_2, \ldots, G_t$ (one group per thread)
Locally shuffle the data for each of the t groups, $G_1, G_2, \ldots, G_t$
for epoch $e = 1$ to i **do**
    for each thread $k = 1$ to t **do**
        Iterate sequentially through all data points in G_k, to obtain gradient estimate
    end for
    Compute the average gradient estimate from the t threads
    Update parameter β_e using the averaged gradient
end for
Output: Final parameter β_i

While shuffling large datasets is itself computationally expensive, this approach significantly speeds up the process by eliminating the need to randomly select each point during training. However, this procedure deviates from the basic assumption of random sampling, as once a point is selected in an epoch, it cannot be selected again—effectively resulting in sampling without replacement. However, the paper concluded that "the convergence rate for random shuffling is comparable to that for i.i.d sampling" [4] after an in depth analysis in both convex and nonconvex cases. We will therefore implement a parallelised version of SGD inspired by this algorithm which we will refer to as **SGD parallelism by Epoch**.

Chapter 2

Experiments

2.1 Preliminary Setup

2.1.1 Data

We chose to evaluate SGD in the context of polynomial linear regression (2.1). This is because the optimization function in linear regression is a convex function, ensuring that minimum estimates are the global solutions to the problem. The fact that it's polynomial helps us visually see our fits and also allows us to have more than two weights to estimate. We obtained a regression dataset from University of California Irvine Machine Learning Repository [5]. It consists of 9568 data samples and is from a Combined Cycle Power Plant collected over 6 years (2006-2011). The Power Plant was set to work at full capacity during this time. Features included hourly average ambient variables Temperature (T), Ambient Pressure (AP), Relative Humidity (RH) and Exhaust Vacuum (V) to predict the net hourly electrical energy output (EP) of the plant. However, we chose the Exhaust Vacuum as the regressor variable to predict Temperature as preliminary plots suggested these variables were a good candidate for a polynomial relationship (as seen in figure 2.1).

Polynomial Regression:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \varepsilon \tag{2.1}$$

where β are the coefficients to be estimated.

2.1.2 Baseline Results

We initially transformed the data using polynomial basis functions with degree 3. We then estimated the weights through Ordinary Least Squares (OLS) and calculated the MSE to obtain a baseline of what Gradient Descent will converge to. We also calculated the MSE of Batch Gradient Descent with $\eta = 0.00001$ for 5000 iterations to see if our baseline gradient descent model did indeed converge to the OLS MSE. We obtained the following preliminary results with their corresponding visual fits (Figure 2.1):

- OLS MSE: 0.28748
- OLS MSE (Polynomial): 0.27182
- Vanilla GD MSE (Polynomial): 0.27182

2.2 SGD parallelism through processes

In order to evaluate the costs and benefits of parallelizing SGD through the method implemented by Zinkevich, Weimer, Li, *et al.*, we conducted the following experiment on the aforementioned dataset:

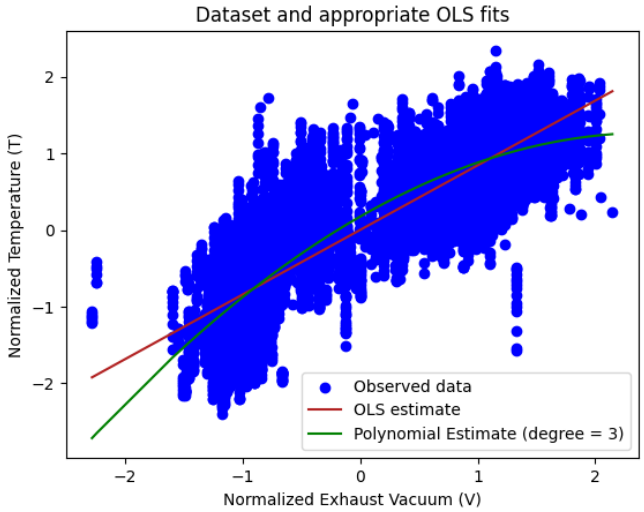


Figure 2.1: Data, and fits through Ordinary Least Squares

2.2.1 Methodology

We created the function `lin_reg_poly_sgd_parallel` which implements algorithm 2 in the context of linear regression. We changed the function so it returned the Mean Square Error (MSE), and time taken to implement the algorithm instead of the parameter weights.

Because of the inherent stochastic nature of the SGD algorithm (caused by random sampling), assuming the same initial random weights for the parameters β, two different runs of algorithm 2 will return two different parameter estimates after T iterations. This is not the case for Batch Gradient Descent. Hence a single run of algorithm 2 is insufficient to understand its behaviour after T iterations. As a result we create the function `get_mse_mean_var_parallelSGD`. We wanted to see the characteristics of our estimates $\hat{\beta}$ after T iterations. Let $\hat{\beta}_T$ be a random variable denoting an estimate from algorithm 2 after T iterations. As a result the MSE is also a random variable after T iterations. This function would call `lin_reg_poly_sgd_parallel` i amount of times, and estimate the unbiased mean and variance of the mean squared errors i.e. $\overline{MSE}$ and $\widehat{Var}(MSE)$. The function also returns the average time taken to execute algorithm 2. The function is depicted by algorithm 5:

Algorithm 5 MeanVarParallelSGD$(\{c^1, ..., c^m\}, T, \eta, \omega_0, k, i)$

```
Initialise list MSEs
Initialise list times
for j ∈ {1, ..., i} do
    mse_j, time_j = SGDParallel({c^1, ..., c^m}, T, η, ω_0, k)
    Append mse_j to MSEs, time_j to times
end for
return MSEs_bar, Var_hat(MSEs), times_bar
```

Ideally, as we increase the number of processes k in algorithm 2, we desire the MSE to reduce on average, and the variance of the MSE to reduce on average for a given number of iterations T. We also desire the amount of time taken to execute algorithm 2 to remain constant as we increase k. As it has been discussed that SGD converges to the minimum achieved by Vanilla GD, we do not concern about convergence results as algorithm 2 will have the same convergence results (and is shown by Zinkevich, Weimer, Li, *et al.* [2]).

We are more concerned whether parallelisation allows us to achieve a better model given restricted time constraints (will it provide us a lower MSE on average for the same T).

We plot $\overline{MSEs}$ as returned by algorithm 5 for $T = 10, 100, 500, 1000, 2000, 3000, 4000, 5000, 7000, 9000$. We repeat this for different process counts $k = 1, 2, 4, 8$. Note that when $k = 1$, this is effectively SGD without parallelisation. We also repeat this for the variances and for the average times.

2.2.2 Results

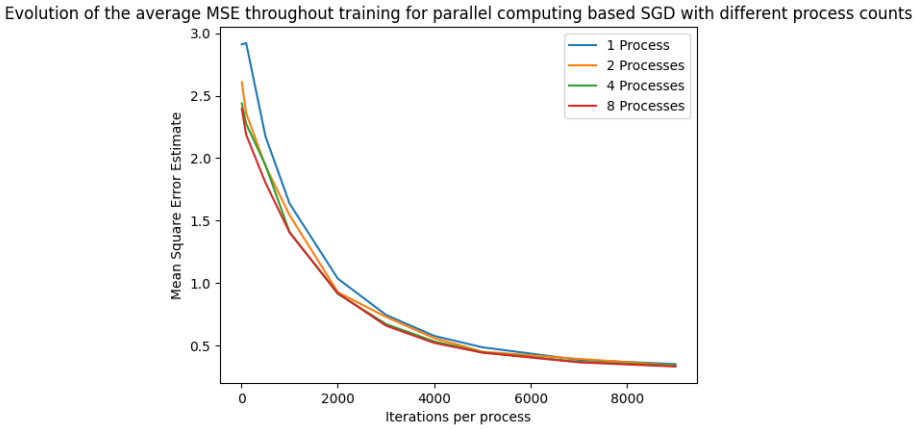
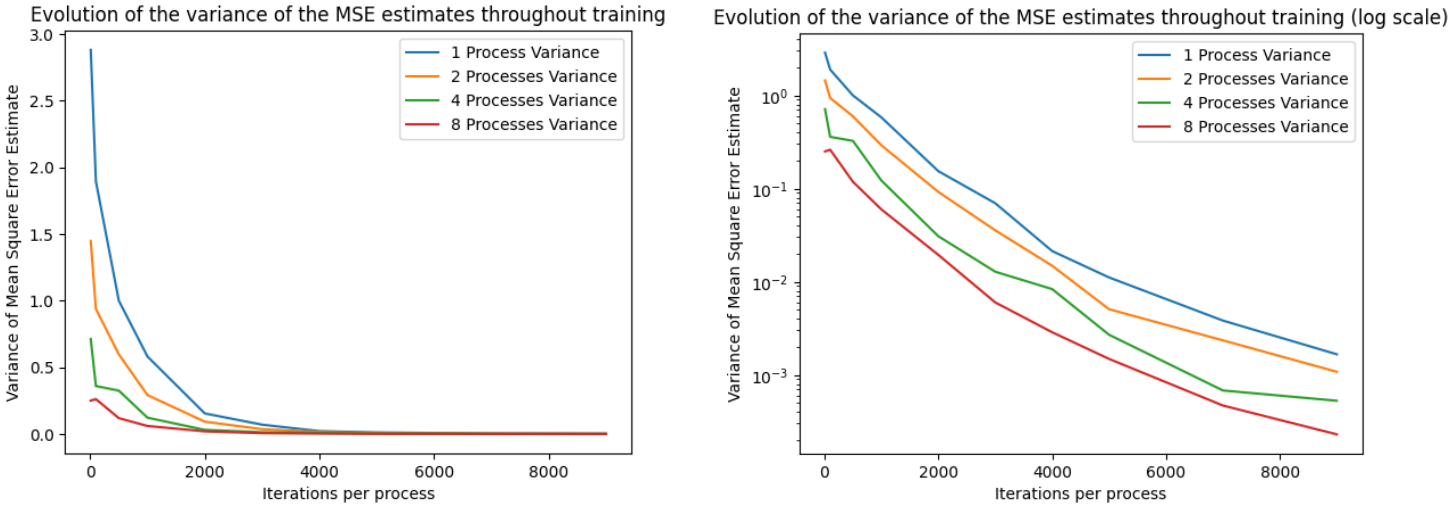


Figure 2.2: Comparing how the average MSE evolves throughout training for different process counts

Throughout all k and T, we use a learning rate $\eta = 0.0001$, and set the number of MSE/time estimates $i = 100$. The weights ω_0 are randomly initialised.

Figure 2.2 depicts our results for plotting $\overline{MSEs}$. We can see that for a given number of iterations, the Mean Square Error does indeed on average decrease as you increase the number of processes! This in theory tells us that we can achieve a better model with the use of parallelism if we are restricted by time (usually in the cases of extremely large datasets). The plot also demonstrates diminishing returns in the benefits of parallelism as the process count k increases. This is because we can see the difference between blue and orange lines (1 and 2 processes respectively) are much larger than the green and red lines (4 and 8 processes respectively).

Figure 2.3(a) and 2.3(b) depict our results of the evolution of the variance of the MSEs. It is expected that the variance of the MSE reduces as you increase the number of iterations for any SGD model in linear regression. This is because the weights in SGD converge to the global minimum in the context of linear regression (due to the convexity of the optimization function). As all the weights converge to similar values, the estimate variances and hence the MSE variances gets smaller. The real insight this plot provides can be viewed in the logarithmic plot. We can clearly see how increasing the number of processes clearly results in more precise estimates from the lower variance for a given iteration count. The benefits appear to empirically increase on a logarithmic scale as you increase the process count, but much further testing is needed to verify this.



(a) Linear Scale

(b) Log Scale

Figure 2.3: Comparing how the variance of the MSEs evolves throughout training for different process counts.

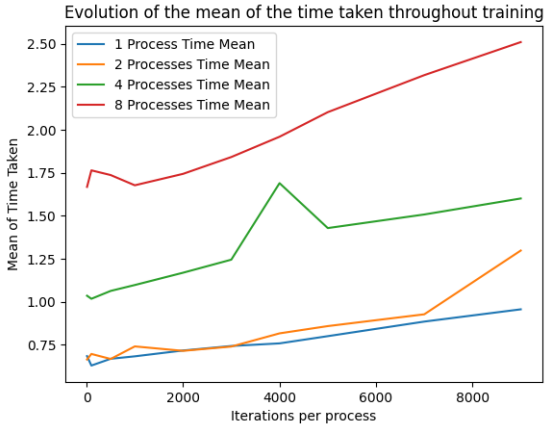


Figure 2.4: Average time taken (seconds) to execute a certain iteration count for different number of processes

However the above benefits don't come without drawbacks. Figure 2.4 shows us the time taken on average to execute each instance of algorithm 2 for a specified number of iterations and processes. It is expected that the line will trend upwards as the algorithm is sequential in nature, and this is observed in all lines. The real drawback can be seen in the huge difference in the average time taken for all iterations between $k = 1$ and $k = 8$. We can see it takes the CPU 1.7 seconds just to start the pool of 8 process in the program (inferred from looking at the 0 iteration count). The red line also appears to be slightly steeper than the blue line, meaning the rate of time increase as you increase iterations is more for a higher k. We hypothesize this is because of the increased load on the system scheduler, which now has to allocate CPU resources between more processes, and also the increased time spent context-switching by the scheduler.

2.2.3 Conclusion

The algorithm proposed by Zinkevich, Weimer, Li, *et al.* has similar benefits in the setting of processes as in the setting of different machines i.e. the benefits of better estimates on average and lower variance for a given iteration count. Implementing the algorithm in the context of processes has a clear drawback which was not observed by them which is the evident increase in wall-clock time. In our estimation of the time

taken, we include the amount of time taken by the program to initialise the pool of k processes, and hence see the clear difference. With the limited amount of data available, we can also see the rate of time increase is higher for a higher k, but further testing is needed with much higher process counts to verify preliminary results.

Finally, the true benefits of parallelizing cannot be truly appreciated in our experiment due to the size of our dataset. Zinkevich, Weimer, Li, *et al.* used a much larger proprietary dataset to see much clearer differences than what could be seen in figure 2.2, with lower machine counts requiring 100,000s of more iterations to achieve the same level of MSE of higher process counts.

2.3 SGD parallelism using k samples

The experiment below uses algorithm 3, proposed by Santos and Ojha. In this algorithm, we sample multiple observations for each iteration in an update (mini-batch gradient descent). We have added shuffling of indices to this algorithm to increase randomness. By varying the number of threads, we will be able to observe how introducing parallelisation affects the convergence rate and efficiency of stochastic gradient descent. Using different values of k will show us how batch size interacts with thread count and affect the algorithm performance. We have chosen to use the loss function in equation 2.2:

$$L(\alpha) = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{2.2}$$

Where m is the number of samples, y_i is the true value, and $\hat{y}_i$ is the predicted value

Therefore the gradient for the loss function was computed as given in equation 2.3.

$$\nabla L(\alpha) = \frac{1}{m}X_k^T(X_k\alpha - y) \tag{2.3}$$

Where X_k is the matrix of size $(k * 4)$ (considering k samples instead of the entire dataset).

2.3.1 Methodology

We created the function `compute_gradient` that accepts the arguments X, y and α (parameter). It transforms X using polynomial basis function with degree 3. This function goes on to make predictions and returns the gradient. The main function `parallel_sgd_with_k_samples` implements algorithm 3 and returns 2 lists containing MSE and time taken (cumulative) for each iteration. Finally to obtain the graph of MSE against time we used an `estimation` function which runs the main function `parallel_sgd_with_k_samples` multiple times and returns 2 lists of averages of MSE and time over the multiple iterations. This was done to ensure the graph obtained are consistent despite the randomness in sampling. (As explained in 2.2.1, MSE is a random variable)

In our script, the following formula is used to compute stochastic gradient descent with k-samples. This implementation allows for parallel computation of gradients across multiple mini-batches, which are then averaged in the main SGD loop.

$$\alpha = \alpha - \eta \cdot \frac{1}{k}\sum_{i=1}^{k}\nabla L(\alpha, x_i, y_i) \tag{2.4}$$

Where:

- α represents the model parameters
- η is the learning rate
- k is the mini-batch size
- $\nabla L(\alpha, x_i, y_i)$ is the gradient of the loss function for a single sample

Note: If we split the dataset such that every datapoint is used i.e. $k * num\_of\_threads (\neq 1) = len(X)$, this algorithm mimics parallelizing batch gradient descent

In this experiment, we have two values that can be varied - k (number of samples per batch) and $num\_of\_threads$ (number of threads used for parallel computing). Ideally we'd like to see that increasing the number of threads (i.e. parallel workers) would lead to a faster convergence to the global minima as would sampling more points (increasing k) each iteration.

2.3.2 Results

To understand how the impact of both k and $num\_of\_threads$, we created 3 graphs of MSE against time taken (in seconds). In each graph we keep the $num\_of\_threads$ constant and show the change in MSE for different values of k. We kept the number of iterations (1000) and the learning rate,$\eta = 0.01$ constant for both graphs and `estimation` ran `parallel_sgd_with_k_samples` for a 100 times(to average the MSEs).

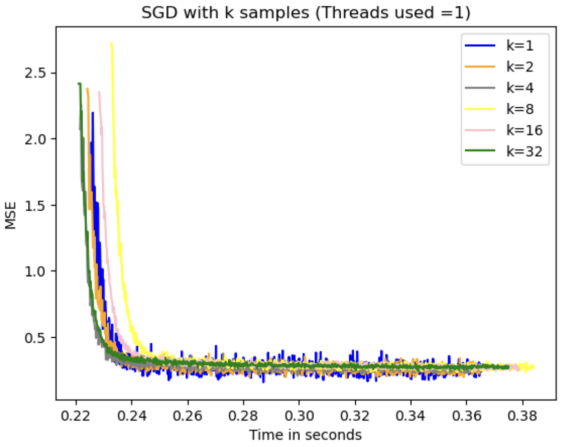
Figure 2.5(a) depicts our results when $num\_of\_threads = 1$. We see here that the MSE decreases (or converges to 0) at around the same rate regardless of the value of k. This is expected as more time will be taken compute the gradient of 32 points than 1 point. While more time is needed to run 1 iteration where $k = 32$, more iterations(or updates) are needed for $k = 1$. It is due to this that Santos and Ojha say "the cost of additional information outweighs the benefits". However the curve appears to be smoother (or have less variance)as the value of k increases.

Figure 2.5(b) depicts our results when $num\_of\_threads = 4$. Here too we see a similar trend as above in regards to the rate of convergence depending on the value of k. No matter what the value of k, MSE converges to 0 at the same rate and there is less variance the greater the value of k. Here we see that adding parallelization appears to slow down the convergence of the MSE by around a tenth of a second. However on running the code multiple times (with different inital values for the parameter), we saw that while the addition of parallelisation slows down the convergence, the value of k does not appear to make much of a difference as each time the MSE converged the fastest for a different k. We then considered additional parallelisation.

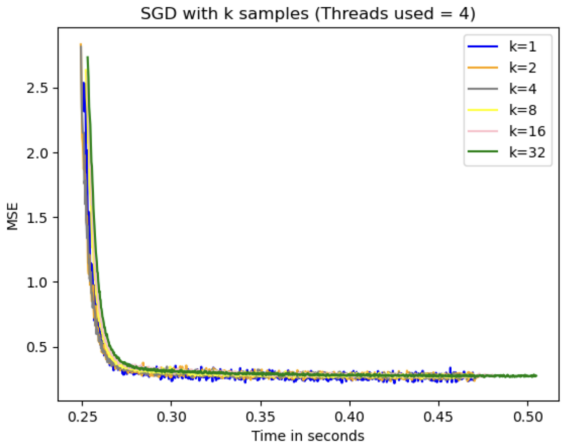
Figure 2.5(c) depicts our results when $num\_of\_threads = 16$. Here too we see that the rate of convergence of MSE is similar no matter what the value of k is and again there is less variance the greater the value of k. Here too we see that adding parallelization slows down the convergence of the MSE by longer than when we used 4 threads. As the number of threads increase, more time is spent on communication and synchronisation(averaging the gradient) also known as communication overhead. As discussed in [6], simply introducing parallelism does not lead to faster performance as most parallel programs require a certain amount of sequential processing. Thus in certain cases like ours, parallelisation is not beneficial i.e. it leads to the convergence slowing down.

2.3.3 Conclusion

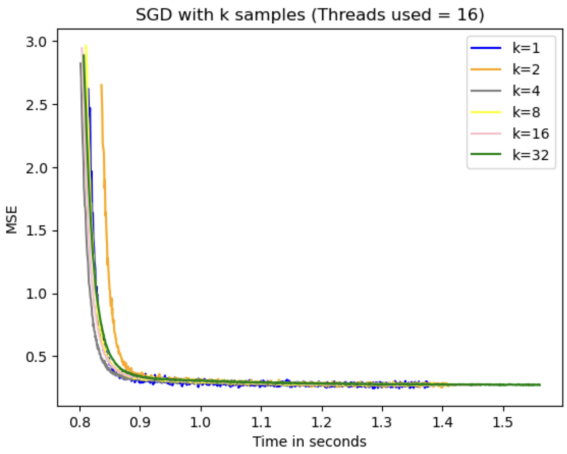
The algorithm, SGD with K samples proposed by Santos and Ojha does not benefit from parallelisation when tested with the dataset we have chosen as the convergence of MSE to 0 takes longer. Contrary to what



(a) Using 1 thread (No parallelization)



(b) Using 4 threads (Parallelization)



(c) Using 16 threads (Parallelization)

Figure 2.5: Comparing the speed of convergence of MSE for different numbers of threads.

was observed in the paper, a small number of additional processors did not accelerate the convergence rate. This difference could be due the choice of dataset. In the paper, two datasets were mentioned. We chose to work with the dataset explained in 2.1.1. The second dataset mentioned in [3] was larger (≈ 40000) which could attribute to the difference. Choice of k is not significantly important in the context of time as the rate of convergence is dependent on the initial parameter which is randomly chosen and based on this dataset, we see that no matter what the value of k is, the difference in time taken to converge is less than a second. However in all three graphs we see that a larger value of k leads to a more stable model (less variance in the MSE vs time curve). Thus we see that there is a tradeoff between model stability and computation time of each iteration. In our experiment, we are not considering the computation time of each iteration but the time taken for the MSE to converge to 0, thus we see that a higher value of k would be better for this algorithm in view of model stability as convergence time is not significantly impacted. However, experimentation with a larger dataset may help finalise an optimum choice of k. Further exploration must be done in regards to both parallelization and choice of k with larger datasets.

2.4 SGD parallelism by Epoch

Meng, Chen, Wang, *et al.* demonstrated that global shuffling (shuffling the entire dataset) results in a faster convergence rate compared to local shuffling (shuffling within different groups of the partitioned dataset)[4]. This advantage is attributed to the information loss inherent in local shuffling, which arises due to the lack of communication between the partitioned data. Given that our dataset is not particularly large, we decided against partitioning the dataset across processors. Instead, each processor is provided with the entire dataset, performs a single shuffle, and iterates through its shuffled version of the dataset for the required number of epochs. Below, we outline the methodology for our parallelized version of SGD, which is inspired by the approach discussed by Meng, Chen, Wang, *et al.*

2.4.1 Methodology

The SGD by epoch approach implements the following steps:

1. **Data Pre-processing**: The input feature X and target variable y were normalized to ensure consistent scaling, which improves the stability and efficiency of model training.
2. **Polynomial Basis Function Transformation**:The input feature X was transformed into polynomial features of degree $h-1$, enabling the model to fit higher-order relationships in the data.
3. **Independent Shuffling of Input Features**: To improve computational efficiency, the data was shuffled independently across t processors at the start of training. Each processor received its own shuffled version of the data, allowing parallel computation without the need for additional shuffling during iterations.
4. **Random Initialization of Weights**: The model weights β were randomly initialized and shared across all processors as the starting point for training.
5. **Processor-specific Epoch Iterations**: Each processor independently iterated through its version of the shuffled data in sequence, for one epoch, updating the weights.
6. **Averaging Results**: After completing each epoch, the t processors returned their estimates of β. These estimates were averaged to obtain a unified global estimate, which was then shared among all processors for the next epoch.

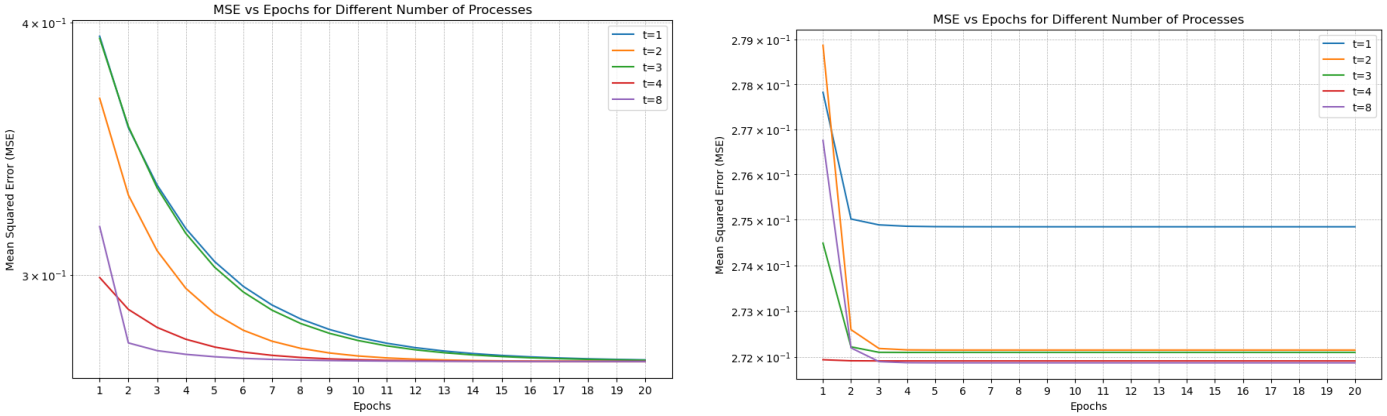
7. **Iterative Updates Across Epochs**: Steps 5 and 6 were repeated for n epochs. The model refined its weights incrementally with each iteration.

8. **Dynamic Learning Rate**: [7] A dynamic learning rate mechanism was introduced to improve stability and convergence during training. The learning rate decreases with the epoch to prevent overshooting during convergence.

9. **Final Weight Extraction**: At the conclusion of training, the final β values represented the optimized weights of the model.

2.4.2 Analysis of Graphs

Before Dynamic Learning Rate Implementation



(a) MSE vs Epochs with a learning rate of 0.0001

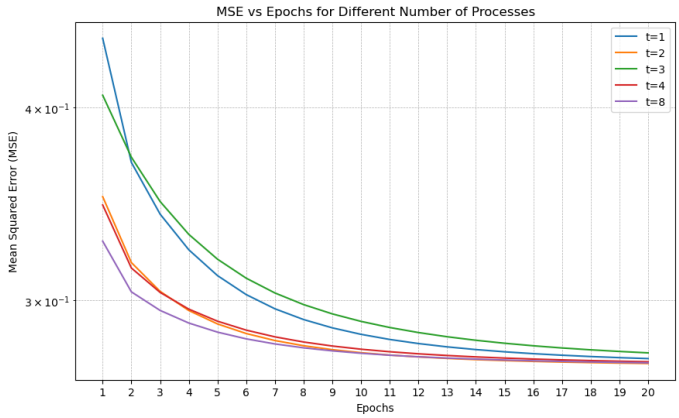
(b) MSE vs Epochs with a learning rate of 0.001

Figure 2.6: Comparison of MSE vs Epochs with different learning rates.

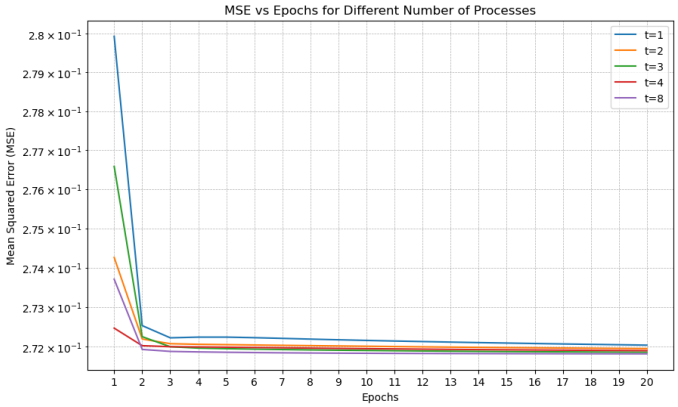
The graphs above illustrate the evolution of the mean squared error (MSE) of our estimates after each epoch. 2.6(a) displays the results with a learning rate of 0.0001, while 2.6(b) shows the results with a learning rate of 0.001. In 2.6(a), we observe that the algorithm exhibits stable convergence with a learning rate of 0.0001. This stability is achieved even though the same shuffled sequences are reused for each epoch, rather than selecting data points randomly for every iteration. The results also highlight the benefits of parallelization as on average the more processes used, the lower the MSE was after each epoch. When using a single processor, the algorithm requires the most epochs to converge. As the number of processors increases from t=3 to t=4, convergence becomes faster in terms of the number of epochs. This pattern repeats itself again from t=4 to t=8 although the change is less noticeable. This plateau in efficiency is likely because the dataset, consisting of 9568 data points, is not large enough to fully utilize the added processors. Although each processor processes all 9568 data points per epoch, convergence (as shown in Figure 2.6(a)) still requires multiple epochs for all processor counts. This is because the small learning rate of 0.0001 leads to slow but stable updates. To investigate the effect of a larger learning rate, we increased it to 0.001, as shown in 2.6(b). With this higher learning rate, convergence occurs much faster, requiring only around three epochs. However, the larger learning rate causes the algorithm to overshoot the optimal solution, leading to inconsistent convergence. This is evident from the slight variations in the final MSE across different runs. In 2.6(b), parallelization again accelerates convergence, as the MSE when using more processors is again generally lower for a given number of epochs compared to fewer processors. However, the overshooting behaviour due to the larger learning rate reduces the stability of convergence. To balance faster convergence with consistency

and stability, we decided to implement a dynamic learning rate in our algorithm. This approach adjusts the learning rate over time, aiming to achieve a more efficient and reliable optimization process.

After Dynamic Learning Rate Implementation



(a) MSE vs Epochs with dynamic learning rate and base learning rate 0.0001



(b) MSE vs Epochs with dynamic learning rate and base learning rate 0.001

Figure 2.7: Comparison of MSE vs Epochs with dynamic learning rate and different base learning rates.

In 2.7(a), the results show the application of a dynamic learning rate with a base learning rate of 0.0001. Compared to a higher base learning rate, the updates are now more cautious, leading to smoother and more stable convergence. The initial progress in reducing the Mean Squared Error (MSE) is slower, as expected, because the smaller base learning rate minimizes the risk of overshooting the optimal solution. This stability ensures consistent progress across epochs. Parallelization continues to play a crucial role, as configurations with more processors (e.g., t=4 and t=8) achieve faster MSE reduction during the early epochs compared to fewer processors. However, after around 10 epochs, the differences between configurations diminish, and all stabilize at similar MSE values. This highlights that while parallelization accelerates early-stage learning, its benefits level off as the algorithm converges. In 2.7(b), a dynamic learning rate is applied while maintaining a higher base learning rate of 0.001. The dynamic adjustment allows the algorithm to achieve significant reductions in MSE during the first few epochs, demonstrating much faster initial convergence compared to 2.7(a). This is because the higher base learning rate makes larger updates in the beginning, quickly approaching lower MSE values. However, the rapid updates may introduce some instability, particularly in the transition to finer adjustments during later epochs. Parallelization proves beneficial in this scenario as

well, with configurations using t=4 or t=8 processors achieving faster reductions in MSE than t=1. After about 10 epochs, all configurations stabilize at similar MSE values, indicating that the dynamic learning rate effectively balances the trade-off between rapid initial learning and later stability. The updated results demonstrate that a strategic combination of a dynamic learning rate and parallelization significantly enhances the efficiency and stability of the Stochastic Gradient Descent (SGD) algorithm. By carefully tuning the base learning rate and leveraging parallel processing, this approach optimizes convergence speed and stability, making it particularly effective for large datasets where computational efficiency is critical.

Comparison and Insights

The dynamic learning rate accelerates convergence during the initial epochs compared to a fixed learning rate. A higher base learning rate (0.001, 2.7(b)) leads to faster initial progress but may risk instability in convergence. A lower base learning rate (0.0001, 2.7(a)) provides more stability but takes longer for the initial reduction in MSE. Parallelization consistently reduces the MSE faster for all configurations, especially in the first few epochs, but its impact diminishes as the algorithm converges. The combination of a dynamic learning rate and appropriate parallelization ensures both faster convergence in the early epochs and stable optimization over time. This demonstrates how carefully tuning the learning rate and leveraging parallel processing can significantly improve the efficiency and effectiveness of optimization algorithms like SGD.

2.4.3 Conclusion

This demonstrates the impact of implementing various optimizations in the Stochastic Gradient Descent (SGD) algorithm to improve convergence efficiency and stability. By introducing parallelization and dynamic learning rates, we successfully balanced the trade-offs between computational speed and model stability. Key insights include the benefits of shuffling data for parallel processing, the effectiveness of dynamic learning rates in reducing Mean Squared Error (MSE) during early epochs, and the role of fine tuning the base learning rate to balance fast initial progress and consistent optimization. The results highlight that a higher base learning rate combined with a dynamic adjustment accelerates convergence significantly but may risk instability if not controlled. On the other hand, a lower base learning rate with dynamic learning provides smoother convergence but requires more epochs to achieve similar progress. Parallelization consistently improved the speed of convergence, particularly in the early stages. However, its efficiency plateaued for larger numbers of processes. Overall, this experiment emphasizes that a strategic combination of dynamic learning rates and parallel processing can significantly enhance the efficiency and effectiveness of the SGD algorithm. These optimizations are particularly valuable when working with large datasets, where computational efficiency and stability are critical to achieving high-quality results. Future work could explore adaptive strategies for determining the optimal base learning rate and scaling parallelization for even larger datasets.

Chapter 3

Conclusion

The introduction of parallelization in Stochastic Gradient Descent proved to be beneficial in 2 out of the 3 experiments performed. However, it was seen that these benefits sometimes come at a cost.

The key insights derived from the 3 experiments are as follows:

1. In the case of the algorithm proposed by Zinkevich, Weimer, Li, *et al.*, while parallelization by processes led to better estimates on average and lower variance for a given iteration count, there is an increase in time taken to implement the algorithm due to the process initialization overhead. We see that the rate of time increase is greater for a larger number of processes indicating that further testing is required with higher processor counts and a larger dataset.

2. In the experiment, SGD with k samples derived from the algorithm discussed in [3], we saw that parallelization was not beneficial due to the communication overhead required to average the gradients computed by the multiple threads. However, the choice of k while not having a significant impact on convergence time, did impact model stability telling us that higher the value of k, the more stable the model would be. While this experiment did not give us our desired results in terms of parallelization, we concluded that mini batch gradient descent would be an improvement on stochastic gradient descent (using $k \neq 1$ to compute the gradient for each iteration).

3. In the experiment, SGD by epochs introducing parallelization along with dynamic learning rates improved both convergence efficiency and model stability. However, it was found that the efficiency plateaued after a certain number of processors, thus indicating that while parallelization was beneficial, a choice of number of processors must be made based on the dataset. This experiment also showed us that careful choices must be made for all parameters such base learning rate and number of processors keeping both the given dataset and other constraints such as time and stability tolerance of the model in mind.

This project highlights that optimizing SGD requires careful consideration of dataset size, parallelization overhead and hyperparameter tuning bearing in mind the tradeoff between computational efficiency and model stability. While we have seen that parallelization can be beneficial, further experiments must be conducted with larger datasets to truly understand the true benefits and formalize strategies for the choice of hyperparameters.

References

[1] S. Ruder, *An overview of gradient descent optimization algorithms*, 2017. arXiv: 1609.04747 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1609.04747.

[2] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23, Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf.

[3] K. Santos and S. Ojha, "Parallelizing gradient descent," Class Project Report for 15-418/15-618, Carnegie Mellon University, 2018. [Online]. Available: https://shashank-ojha.github.io/ParallelGradientDescent/Final%20Report.pdf.

[4] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2019. DOI: https://doi.org/10.1016/j.neucom.2019.01.037.

[5] P. Tfekci and H. Kaya, *Combined Cycle Power Plant*, UCI Machine Learning Repository, 2014. DOI: https://doi.org/10.24432/C5002N.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 1967, pp. 483–485.

[7] Towards Data Science, *Learning rate schedules and adaptive learning rate methods for deep learning*, https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1, Accessed: January 27, 2025.