THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF ELECTRICAL ENGINEERING

ERROR CORRECTING CODES FOR DISTRIBUTED NON-LINEAR FUNCTION COMPUTATION

THITIWAT TAPHA
Spring 2024

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Electrical Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

Dr. Viveck Cadambe
Associate Professor of Electrical Engineering
Thesis Supervisor

Dr. Julio Urbina
Professor of Electrical Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

We provide novel coded computational strategies for distributed non-linear function computation. We utilize the idea of previous studies on coded matrix-multiplication computation and extend our study to achieve optimal error in non-linear distributed logistic regression. We consider the system of P computation nodes where each node receives two one-dimension input matrices. Our computation system reduces the error from stragglers via encoding and decoding. Our experiment with a random set of the number of worker nodes, threshold, and the number of inputs has yielded conclusions on the effect of parameters on the loss function. Moreover, we develop an optimization framework based on alternating minimization that enables the discovery of new codes to achieve optimal coefficients to minimize the error in non-linear functions.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This thesis acknowledgment is a tribute to all the people who made my academic journey worthwhile. I would like to express my deepest gratitude and appreciation to the many individuals who have supported and inspired me throughout my academic journey. Their guidance, encouragement, and unwavering belief in my abilities have been significant in shaping my research and personal development.

I would like to begin by thanking my thesis supervisor, Professor Viveck Cadambe, who provided me invaluable expertise and guidance. He has given me opportunities to grow as an engineering student, a researcher, and also as a person. I feel grateful for all of the opportunities in my life he has given me and all of the times that he has dedicated to me and this research. I also am thankful for his patience, encouragement, support, and kindness. I also would like to thank Dr. Julio Urbina, my honor advisor, for his time and support in making sure that I received the best academic experience possible as an honors scholar.

I would like to extend a personal note of gratitude to my family and friends who have supported me along this journey. I want to say special thank you to one of my best friends, Sorawit Mattra, who spent countless days and nights with me while I worked on this thesis and suggested to me while I wrote my codes. I also. want to say thank you to my graduated friends, particularly Pornpat, Jidapa, Ruthirut, and Supitchaya who have helped my younger self to pass through my academic and life difficulties. I also want to say thank you to all of the friends, I have made along this academic journey. Without all of them, my journey wouldn't be this colorful. I want to say thank you to my parents, though they couldn't be here with me, they always did anything they could to help and support me.

Lastly, I want to say thank you to my younger self who never stop challenging himself since the day he decided to come to the United States, to Penn State, and to join Schreyer Honor College. I want to say thank to his determination and afford to improve himself. Without him, I couldn't be a strong person as I am now.

To all those who have contributed to my academic journey, whether through guidance, support, or inspiration, I offer my heartfelt thanks. Your impact on my life and work will not be forgotten.

Chapter 1

Introduction

In modern computing, computation is done in parallel and large-scale distributed systems. However, even a few slow computing nodes, known as *stragglers*, can significantly slow down the overall computation. As such, the performance of distributed computing systems is limited by straggling nodes in the system. Stragglers, which are potentially caused by the nodes being prone to errors and delays, can produce unpredictability and unreliability in distributed computing systems [2][3]. Finding solutions to solve the problem due to stragglers and enhancing the efficiency of parallelization remains an interesting research challenge in the field of high-performance computing [4][5][6].

Coded computing is an emerging sub-field within the information theory research that focuses on developing techniques to improve the efficiency, reliability, and security of distributed computing systems [7][8, 1]. In particular, coded computing has developed several techniques to add redundancy to distributed computing to ensure that the overall computation can proceed even if there is a limited number of stragglers in the system. Previous work in coded computing has made significant progress in handling linear operations, such as matrix-vector, matrix-matrix multiplications, and polynomial operations [7]. Notable coded computing techniques for coded matrix-multiplication computation include MatDot codes and Polynomial codes, both of which outperformed traditional and Algorithm-Based Fault Tolerance (ABFT) literature[9] in terms of *recovery threshold*, the minimum number of successful nodes required to achieve the computation.[1] However, handling non-linear functions remains an open question in coded computing. Non-linear operations are prevalent in many computational tasks, particularly in machine learning and scientific computing, where functions can be complex and highly non-linear. This thesis aims to make progress by developing new coded computing techniques that can handle non-linear functions.

In this study, our focus is on logistic regression, a statistical technique utilized for binary classification and decision-making purposes[10]. Logistic regression is a tool for predicting the likelihood of finite outcomes, typically represented by binary results. As such, it stands as a fundamental and ubiquitous technique in machine learning as it uses this function to predict the outcome on a provided set of multiple independent variables [11]. In simple form, given a $N \times 1$ data vector $\vec{x}$, logistic regression is parametrized by a $N \times 1$ weight vector $\vec{w}$ and outputs:

$$f_{\vec{w}}(\vec{x}) = \frac{1}{1 + e^{-k\vec{w}^T\vec{x}}}$$

This statistical technique finds widespread use in forecasting the likelihood of finite outcomes, often exemplified by binary results, such as yes or no, because of its range from zero to one. Its capability to predict outcomes plays an important role in decision-making in various fields such as machine learning. As machine learning becomes a more important technology nowadays, logistic regression becomes a more interesting non-linear function in coded computing[12].

In this study, as a starting point, we consider using the system model from MatDot code [1] and aim to imitate the strategy to achieve the optimal error in our non-linear distributed logistic regression. Due to the complexity of non-linear functions, the strategy used to optimize error for linear matrix multiplication proposed in the MatDot code via explicit analytical code constructions is not tractable. Instead, we will employ a novel approach via finding code constructions through optimization. Specifically, we centrally cast the code construction as an optimization problem and use gradient descent - an iterative optimization technique that minimizes a function by computing its gradient - to solve the optimization problem. In particular, the optimization problem will

be solved via an alternating optimization algorithm to achieve locally optimal coefficients - with gradient descent being used as a heuristic for alternating optimization.

A challenge we face in formulating the optimization framework is that, unlike linear functions, the gradient of logistic regression depends on the input variables. We introduce Monte Carlo simulation, the sample randomization method using probability to approximate the occurrence of the event without using mathematical expression, into our strategy. This simulation will provide the possible outputs based on the inputs and the mathematical expression to examine the impact of inputs on the gradient derivative of functions.

1.1 System Model and Problem Statement

In this section, we establish the notation and its definitions that will be used throughout this study, set the foundational framework for our research, and define the problem we aim to address.

1.1.1 Notations and Definitions

In this work, we use the computational system that consists of three types of nodes which include a master node, *P* worker nodes, and a fusion node. The master node distributes the input in a manner to the worker nodes. The worker nodes perform multiplication with lower complexity and send the result to the fusion node. A fusion node computes the output if the satisfied result is received.[13] As previous studies proposed the coded optimization algorithm for linear matrix multiplication, we aim to propose the coded optimization algorithm for non-linear distributed logistic regression.

There are some notations used throughout this thesis.

- *P*: The number of worker nodes used within the system.
- *m*: The dimension of the input matrix.
- *k*: The recovery threshold, the minimum number of successful nodes required for the system to achieve the computation.
- *N*: The number of samples in Monte Carlo simulation.
- A: The $1 \times N$ input matrix.
- B: The $1 \times N$ input matrix.
- $\alpha^{(i)}$: The encoding coefficient for the element in A input matrix.
- $\beta^{(i)}$: The encoding coefficient for the element in B input matrix.
- $d_{(p)}^{(i)}$: The decoding coefficient at the fusion node.

1.1.2 System Model

We consider a computational system that consists of three types of nodes: (i) a master node; (ii) worker nodes; and (iii) a fusion node.

Definition

(i) A master node that receives the input, i.e., matrices A and B, $1 \times m$ matrices.

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_m \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_m \end{bmatrix}$$

(ii) *P* worker nodes that perform the operations of matrix multiplications in logistic regression will generate encoding coefficients $\alpha^{(i)}$ for A and $\beta^{(i)}$ for B, when $i = 1, 2, 3, \ldots, P$

$$\alpha^{(i)} = \begin{bmatrix} \alpha_1^{(i)} \\ \alpha_2^{(i)} \\ \vdots \\ \alpha_m^{(i)} \end{bmatrix} \beta^{(i)} = \begin{bmatrix} \beta_1^{(i)} \\ \beta_2^{(i)} \\ \vdots \\ \beta_m^{(i)} \end{bmatrix}$$

After receiving the inputs from the master node, worker nodes will compute its encoded output by using $\alpha^{(i)}$ and $\beta^{(i)}$ to A and B, respectively. We denote $C^{(i)}$ as the decoded output from any i nodes.

$$C^{(i)} = \frac{1}{1 + e^{-\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B}}$$

The worker nodes send the result to the fusion node if the computation is successful. The worker nodes do not send the result to the fusion node if it fails.

(iii) A fusion node that receives the output from the successful worker nodes. The fusion node will perform the computation if the number of successful worker nodes exceeds the recovery threshold, *k* or declares "computational failure" if otherwise. [1] At the fusion node, the output from successful worker nodes will be decoded by $d_{(p)}^{(i)}$. We denote p as the error pattern index in S_p, the set of successful nodes in the system.

$$C = \sum_{i \in S_p} \frac{d_{(p)}^{(i)}}{1 + e^{-\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B}}$$

1.1.3 Problem Statement

We consider the system including the master, workers, and fusion node computes and evaluate matrix multiplication on logistic regression. The system takes a row matrix, A, and a column matrix, B, as the inputs and computes the logistic regression of the two metrics multiplication.

Since communication systems can have errors due to stragglers, we simulate a computational system that is prone to failure and delay, allowing us to test and develop strategies to recover the

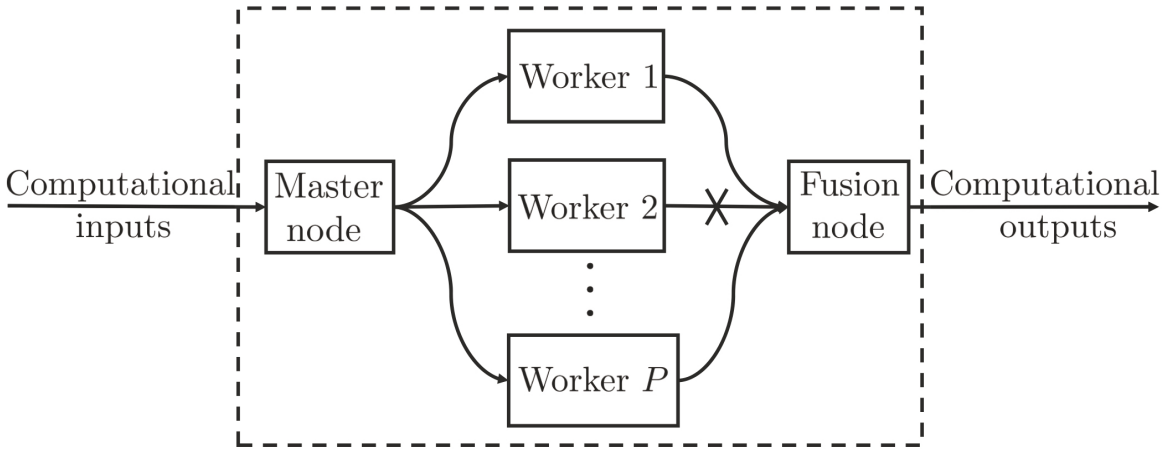

Figure 1.1: A computational system[1]

output from errors.[1][5] Our objective is to find encoding and decoding coefficients for the system to optimize the error caused by stragglers. These coefficients are essential for error correction and data redundancy, which help the system recover from the effects of stragglers. By optimizing these coefficients, the system can prevent the impact of delays and errors, ensuring that the overall system can work efficiently even in the presence of problematic nodes.

In our study, we extend the scope of the MatDot code from its original purpose which is written to achieve a small recovery threshold on matrix multiplication. However, we aim to implement this algorithm on non-linear functions, specifically in logistic regression. We will demonstrate coding strategies to achieve the optimization of the distributed non-linear logistic regression. By enchanting these coding strategies, we have new possibilities for improving predictive models and addressing complex scenarios where linear relationships are insufficient to capture the underlying data patterns.

1.2 Objectives

The objective of this study is to present the coded matrix multiplication, particularly on distributed non-linear logistic regression. This study aims to develop coding strategies to improve the computational efficiency and accuracy of logistic regression models. The objective is to design encoding coefficients αs and βs and decoding coefficients ds to ensure that the mean square error is minimized.

$$min\left(\frac{1}{N}\sum_{m=1}^{N}\left\|\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right\|^2\right)$$

By providing the input parameters *P, m* and *k*, we aim to develop the code that outputs the encoding efficient, α and β, and the decoding efficient, *d*. These coefficients will be used to optimize the error caused by stragglers. Furthermore, this study desires to demonstrate a practical application

of coded matrix multiplication in the context of distributed non-linear logistic regression.

1.3 Related Work and Existing Method

To understand the algorithm, in this section, we review and assess the current approaches and techniques relevant to our study. Using the method that has been previously used in the area of study, we aim to establish an understanding to reach our objectives.

1.3.1 MatDot Code

The MatDot Code is a study on coded computation strategies for distributed matrix-matrix multiplication. The MatDot uses a computational model that consists of master, worker, and fusion nodes. The input matrix is distributed to worker nodes to compute the result and send the output to the fusion node. This novel strategy uses linear combinations of the input matrices to calculate the output. To optimize the error from stragglers, the MatDot Code encodes the encoding coefficients at the master node and decodes the decoding coefficients at the fusion node.

The MatDot demonstrates the ability to minimize the recovery threshold, the minimum number of successful nodes required for the system to compute the output, and the system in several computation systems by lowering the computational complexity at the worker nodes. Thus, the MatDot Code can be used to avoid latency caused by several factors including the limitation of communication system or queuing from the tasks.

1.3.2 Gradient Descent Method

Gradient Descent Method is the common approach used to optimize the variable. The principle of this method is to use slope to find the convex point of the function by iteratively updating the point using the slope at each point. This method is applicable under two primary conditions: (i) function is continuous over the considered interval. (ii) function shows its convexity over the considered interval. The expression for the Gradient Descent can be written as:

$$\Theta_{i+1} = \Theta_i + \gamma \frac{\partial}{\partial \Theta} J(\Theta_i)$$

where: $J(\Theta_i)$ is the loss function.
Θ is the parameter being optimized.
γ is the learning rate.
This method will stop when the system reaches its maximum iteration limit or the gradient of the loss function reaches zero. By choosing the appropriate learning rate, this method can converge to the point of convexity.

In addition, methods that will be used in this topic are **alternating optimization algorithm** [14] and **Monte Carlo simulation**.[15]

Chapter 2

An Optimization approach and Approximate codded computing

In this chapter, we will discuss the strategy of approaching and approximating coded computation. As we are aiming for the minimum error, we propose an optimization method where the loss function between the expected output and the obtained output from the system is minimized. The goal of this optimization is to find the encoding and decoding coefficients.

In section 2.1, we illustrate the optimization framework for the derivation of the formula formally for arbitrary values of parameters *P, k,* and *m*. In section 2.2, we will illustrate the coding strategy used for the Matlab code.

2.1 Optimization Formulation

2.1.1 Randomization Strategy

As mentioned in Chapter 1, the Monte Carlo simulation is commonly used to predict the probability of the event occurrence without using mathematical expressions. In this optimization strategy, the Monte Carlo simulation will be the randomization method to examine the influence of the input matrices on the loss function and its effect on gradient derivatives in this problem. In particular, we will use Monte Carlo to randomize the value of the matrices, *A* and *B*, within the range of zero to one.

2.1.2 Decoding optimization

As mentioned earlier in the previous chapter, the decoding coefficient, *d*, is used to decode the result after the nodes declare successful computation. By taking the partial derivative of the loss function, we found that the derivative of the loss function with respect to *d* has a convex or the minimum. We will solve for the gradient derivative of this loss function and use the gradient descend algorithm to find the minimum point.

In this section, we denote the encoding coefficients following the parameters, *P, m,* and *k* arbitrary. Thus, the loss function can be for *i-th* nodes can be written as:

$$Error = \frac{1}{N}\sum_{m=1}^{N}\left\|\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right\|^2$$

We denote S_p as the set of successful worker nodes. We will demonstrate the optimization formulation of the decoding coefficients. We began with taking the gradient derivative of the coefficients $d_{(i)}^{(p)}$ when i is any of the nodes that respond first and p is the error pattern. To calculate p, we use the binomial coefficient formula $\binom{n}{k}$ For example, $d_{(3)}^{(1)}$ is the decoding coefficient of node 1 in error pattern 3.

$$\frac{\partial}{\partial d_{(p)}^{(i)}}Error = \frac{2}{N}\left(\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\left(\frac{\partial}{\partial d_{(p)}^{(i)}}\left(\sum_{i\in S_p}\frac{1}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\right)\right)$$

However, in any pattern error, there is only one node that corresponds to the following $d_{(i)}^{(p)}$. We can express this using the following expression:

$$\frac{\partial}{\partial d_{(p)}^{(i)}}\left(\sum_{i\in S_p}\frac{1}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\right)=(\frac{1}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}})$$

Therefore, the expression of the gradient derivative of any decoding coefficients can be written as:

$$\frac{\partial}{\partial d_{(p)}^{(i)}}Error=\frac{2}{N}\left(\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\left(\frac{1}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\right)$$

2.1.3 Encoding optimization

In this section, we will discuss the strategy for optimizing the encoding coefficients, α and β. The non-linear relationship between the encoding coefficients and the loss function. Thus, the optimization strategy used in MatDot code and ABFT matrices computation cannot be implemented with these problems. For this specific problem, we found that the plot between the loss function and the coefficients has either convex or minimum points. Thus, we will also use the gradient descent algorithm as the method of optimization.

$$Error=\frac{1}{N}\sum_{m=1}^{N}\left\|\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right\|^2$$

We will demonstrate the optimization formulation of the encoding coefficients. We began with taking the gradient derivative of the coefficients $\alpha_{(\lambda)}^{(\gamma)}$ when γ is any specific node in the system and λ is the input variable. For example, $\alpha_{(1)}^{(3)}$ is the decoding coefficient of node 3 for the input A_1.

$$\frac{\partial Error}{\partial\alpha_{(\lambda)}^{(\gamma)}}=\frac{2}{N}\sum_{m=1}^{N}\frac{\partial}{\partial\alpha_{(\lambda)}^{(\gamma)}}\left(\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)$$

We found that the derivative of the encoding coefficients won't include every term in the error patterns, but only the terms that involve that specific α or β. Therefore,

$$\frac{\partial}{\partial\alpha_{(\lambda)}^{(\gamma)}}\left(\sum_{i\in S_p}\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)=\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\cdot\frac{\partial}{\partial\alpha_{(\lambda)}^{(\gamma)}}\sum_{i\in S_p}(\frac{d_{(i)}^{(p)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}})$$

$$=\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\cdot\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\cdot\frac{\partial}{\partial\alpha_{(\lambda)}^{(\gamma)}}e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}\right)$$

$$=\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\cdot\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\cdot e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}\cdot A\cdot\beta^{(i)\prime}B\right)$$

Thus, we can rewrite the gradient derivative of $\alpha_{(\lambda)}^{(\gamma)}$ as following:

$$\frac{\partial Error}{\partial \alpha_{(\lambda)}^{(\gamma)}} = \frac{2}{N}\sum_{m=1}^{N}\left(\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\cdot\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\cdot e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}\cdot(A\cdot\beta^{(i)\prime}B)\right)\right)$$

Repeating the same process, we can obtain the Mathematical expression of $\beta_{(\lambda)}^{(\gamma)}$ for any specific node (λ) and input variable (γ) which can be written as:

$$\frac{\partial Error}{\partial \beta_{(\lambda)}^{(\gamma)}} = \frac{2}{N}\sum_{m=1}^{N}\left(\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}-\frac{1}{1+e^{-AB}}\right)\cdot\sum_{\substack{i\in S_p\\ i=\gamma\\ j=\lambda}}\left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}}\cdot e^{-(\alpha^{(i)\prime}A\cdot\beta^{(i)\prime}B)}\cdot(\alpha^{(i)\prime}A\cdot B)\right)\right)$$

2.2 Coding Strategy

2.2.1 Coding Strategy for Decoding Coefficients

In this section, we will propose a coding strategy to optimize the decoding and encoding coefficients. We start with randomizing the input variables using Monte Carlo simulation and initializing the encoding and decoding coefficients. Then, we calculate the gradient of the coefficients and use gradient descend to optimize as follows:

Algorithm 1: Decoding Coefficient Minimization Algorithm

Data: $P, m, k, n, N, MaxIterations, LearningRate \geq 0, P \geq k$
Result: $A, B, d_{(i)}^{(p)}$;
Random $m \times P$ matrices α and β;
Random $1 \times N$ array of A and B;
Compute combinations from P and k;
Random $k \times size(combinations, 1)$ matrices d;

Compute matrices $AB = A\dot{B}$;
Compute $Expected = \frac{1}{(1+e^{(-AB)})}$
for $iteration = 1 : MaxIterations$ **do**
  for $i = 1 : P$ **do**
    Compute $N \times P$ matrix $term = \frac{1}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}}$;
    Compute $N \times P$ matrix $expo = e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}$;
  end
  for $i = 1 : size(combinations, 1)$ **do**
    Compute $size(combinations, 1) \times N$ matrix $D = \sum_{i \in S_p} \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}}$;
    Compute $size(combinations, 1) \times N \times k$ matrix $D_i = \frac{1}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}}$;
  end
  for $i = 1 : size(combinations, 1)$ **do**
    Compute $k \times size(combinations, 1)$ matrix
    $gradient = \frac{2}{N} \cdot \left(\sum_{i \in S_p} \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} - \frac{1}{1+e^{-AB}} \right) \left(\frac{1}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} \right)$;
  end
  Compute gradient descend algorithm:
  $d_{i+1} = d_i - LearningRate \times gradient$;
  Compute error: $Error = \frac{1}{N} \sum_{m=1}^{N} \left\| \sum_{i \in S_p} \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} - \frac{1}{1+e^{-AB}} \right\|^2$
end

2.2.2 Coding Strategy for Encoding Coefficients

Following the method of alternating optimization, after optimizing the decoding coefficients, we will optimize the encoding coefficients. In this section, we will demonstrate the optimization algorithm for α since the algorithm for β has a similar procedure as for α. Following a similar setup as the algorithm for decoding coefficients, we process our algorithm as follows:

Algorithm 2: α Optimization Algorithm

Data: $P, m, k, n, P, MaxIterations, LearningRate \geq 0$, $P \geq k$
Result: $A, B, \alpha_{(\lambda)}^{(\gamma)}$;
Random $m \times P$ matrices α and β;
Random $1 \times N$ array of A and B;
Compute combinations from P and k;
Random $k \times size(combinations, 1)$ matrices d;

Compute matrices $AB = A\dot{B}$;
Compute $Expected = \frac{1}{(1+e^{(-AB)})}$

Algorithm 2: α Optimization Algorithm (continued)

for $iteration = 1 : MaxIterations$ **do**

  for $i = 1 : P$ **do**

    Compute $N \times P$ matrix $term = \frac{1}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}}$;

    Compute $N \times P$ matrix $expo = e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}$;

  end

  Compute $P \times m$ matrix $AbB = A \cdot \beta' B$;

  Compute $size(combinations, 1) \times N$ matrix $D = \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}}$;

  Compute $P \times size(combinations, 1)$ setComb array as the array indicates if the given nodes appear in each error pattern.

  Compute $P \times k \times size(combinations, 1)$ martix

  $DForAlpha = \sum_{m=1}^{N} \left(\sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} \right)$ and matrix

  $ExpectedForAlpha = \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \frac{1}{1+e^{-AB}}$

  Compute $P \times N \times size(combinations, 1)$ matrix

  $DExpo = \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} \cdot e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)} \cdot (A \cdot \beta^{(i)\prime} B) \right)$

  Compute $N \times size(combinations, 1)$ matrix $X = \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} - \frac{1}{1+e^{-AB}}$

  Compute $P \times N \times size(combinations, 1)$ matrix

  $DDX = \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} - \frac{1}{1+e^{-AB}} \right) \cdot \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} \cdot e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)} \right)$

  Compute $n \times P \times N$ matrix

  $Y = \left(\sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha'_{(i)} A \cdot \beta'_{(i)} B)}} - \frac{1}{1+e^{-AB}} \right) \cdot \right.$

  $\left. \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} \cdot e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)} \cdot (A \cdot \beta^{(i)\prime} B) \right) \right.$

  for $i = 1 : size(combinations, 1)$ **do**

    Compute $k \times size(combinations, 1)$ matrix

    $gradient = \frac{2}{N} \sum_{m=1}^{N} \left(\sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} - \frac{1}{1+e^{-AB}} \right) \cdot \right.$

    $\left. \sum_{\substack{i \in S_p \\ i=\gamma \\ j=\lambda}} \left(\frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} \cdot e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)} \cdot (A \cdot \beta^{(i)\prime} B) \right) \right.$;

    Compute gradient descend algorithm: $\alpha_{i+1} = \alpha_i - LearningRate \times gradient$;

    Compute error: $Error = \frac{1}{N} \sum_{m=1}^{N} \left\| \sum_{i \in S_p} \frac{d_{(p)}^{(i)}}{1+e^{-(\alpha^{(i)\prime} A \cdot \beta^{(i)\prime} B)}} - \frac{1}{1+e^{-AB}} \right\|^2$

  end

end

Chapter 3

Result and Discussion

In this chapter, we summarize the result of running an optimization algorithm for various combinations of P, k, m in . We report the best result out of 100 random initialization (seeds). For each trial, we ran the Algorithm for 200 iterations. In Figure 3.1, we plot the loss function while running the algorithm on the system of $m = 10, k = 7$, and $P = 8$ using three different seeds. The plot shows different initial values of the loss function when initializing the input variables and coefficients using different seeds. The plot also shows that the algorithm will stop iterating when the loss saturates, which we interpret to be (close to) a local minimum.

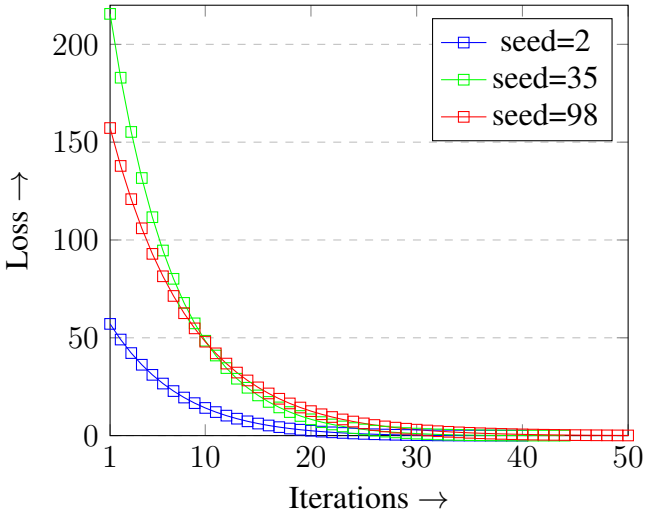

Figure 3.1: Plot of the loss function while running the algorithm with three different seeds

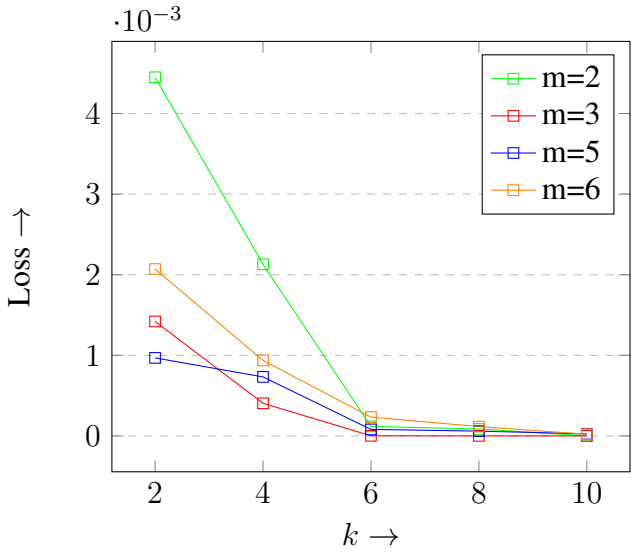

Figure 3.2: Plot of the best result out of 100 initialization at $P = 10$, varying k and m

In Figure 3.2, we plot the minimum loss which is the loss from the best code picked from all initialization. When we vary k for fixed $P = 10$, we observe that as the recovery threshold approaches the number of worker nodes in the system, the loss decreases. In theory, as the recovery threshold increases, fewer stragglers are allowed for the system to complete the computation. We expect better accuracy with fewer stragglers as confirmed by the plot.

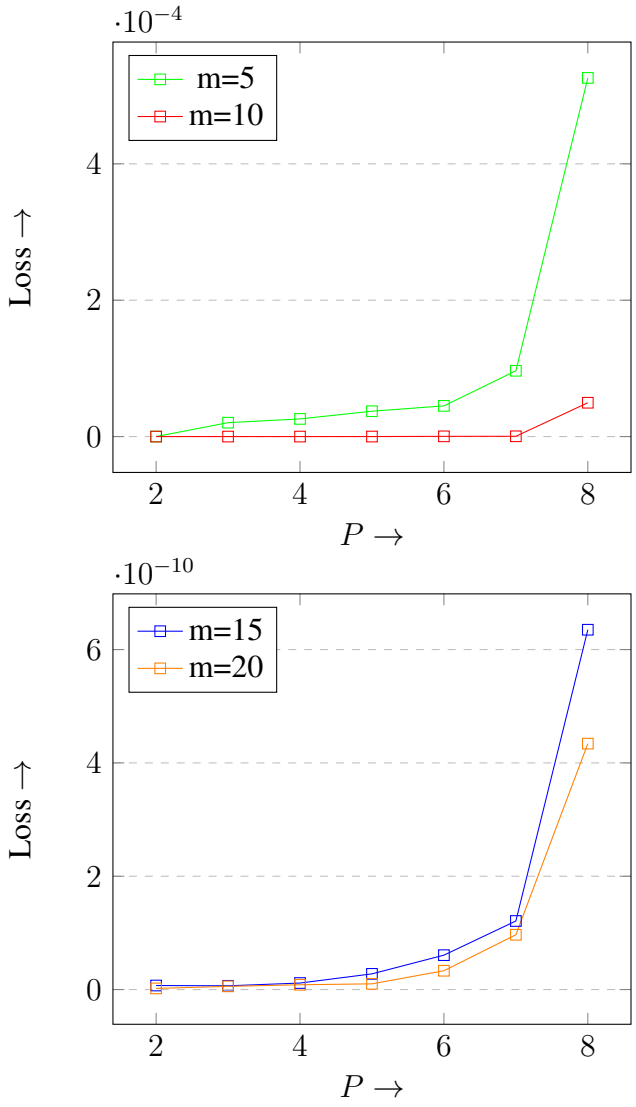

Figure 3.3: Plot of the best result out of 100 initialization at $k = 2$, varying P and m

We also ran the algorithm with larger numbers of input variables. We report the best result out of 100 random initialization (seeds). When varying P for fixed $k = 2$, we observe that the loss function decreases as larger numbers of input variables. In Figure 3.3, the loss function of the system $(20, 8, 8)$ is approximated to be around 4×10^{-10} after running the algorithm.

From our experiment by varying $P, k,$ and , m, the report from the best result out of 100 random initialization shows the effects of the parameter on the loss functions.

(m, k, P)	Error	(P, k, m)	Error
$(200, 4, 4)$	1.2368×10^{-17}	$(10, 18, 20)$	8.24×10^{-6}
$(400, 5, 5)$	1.6539×10^{-16}	$(10, 39, 40)$	9.78×10^{-4}
$(600, 5, 5)$	4.4936×10^{-16}	$(20, 48, 50)$	1.82×10^{-8}
$(800, 6, 6)$	4.0436×10^{-17}	$(20, 60, 60)$	2.05×10^{-13}
$(1000, 7, 7)$	4.0874×10^{-16}	$(30, 75, 75)$	5.42×10^{-26}
$(1100, 8, 8)$	1.2375×10^{-15}	$(30, 25, 25)$	8.07×10^{-21}
$(1200, 8, 8)$	2.2425×10^{-16}	$(30, 23, 25)$	4.31×10^{-20}

Table 3.1: Error of large computation systems

Chapter 4

Conclusions and Future Work

4.1 Conclusion

In this study, we present and demonstrate a novel optimization algorithm used to achieve the optimization of the distributed non-linear logistic regression. Our proposed optimization algorithm begins by utilizing the principles of coded computing particularly based on MatDot code to optimize non-linear logistic regression. The gradient descent algorithm and alternating optimization algorithm are utilized in this study to achieve the optimal value of the decoding and encoding coefficients to minimize the loss function. Additionally, Monte Carlo simulation is employed to represent the input samples.

The experiment with varying $P, k,$ and m shows that our optimization algorithm demonstrates satisfying results. From our experiment with a random set of parameters $P, k,$ and m, we have concluded that, first, the loss function decreases as the threshold, the minimum number of successful nodes required to complete the computation, reaches the number of the worker nodes in the system. Second, we conclude that the loss function is inversely proportional to the number of input variables. In other words, the loss function decreases as the number of input variables increases.

4.2 Future Work

The algorithm proposed in this paper serves as a great initiation for the optimization algorithm for distributed non-linear functions. In this study, we demonstrate our algorithm for distributed non-linear regression by randomizing the input using Monte Carlo simulation. Future work on the extension of this study could be the extensive simulation of real-world data sets. Additionally, the way of splitting the input A and B could be differed by setting A as the set of weights of the classifier, which is available everywhere. Input B can be set as real-world data. The algorithm can only encode B and develop the code construction for this setup. Furthermore, the algorithm can be extended beyond the logistic regression as our approach may be applicable to other non-linear functions. Future work can involve the optimization of the decoding and encoding coefficients of other non-linear functions such as logarithms or exponential functions.

Bibliography

[1] Sanghamitra Dutta, Mohammad Fahim, Farzin Haddadpour, Haewon Jeong, Viveck Cadambe, and Pulkit Grover. On the optimal recovery threshold of coded matrix multiplication. *IEEE Transactions on Information Theory*, 66(1):278–301, 2019.

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, 2013.

[3] Kai Xing, Fang Liu, Xiuzhen Cheng, and David HC Du. Real-time detection of clone attacks in wireless sensor networks. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 3–10. IEEE, 2008.

[4] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 30–46, 2019.

[5] Haewon Jeong, Ateet Devulapalli, Viveck R Cadambe, and Flavio P Calmon. -approximate coded matrix multiplication is nearly twice as efficient as exact multiplication. *IEEE Journal on Selected Areas in Information Theory*, 2(3):845–854, 2021.

[6] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[7] Qian Yu. *Coded computing: A transformative framework for resilient secure private and communication efficient large scale distributed computing*. PhD thesis, Ph. D. dissertation, Dept. Elect. Comput. Eng., University of Southern . . . , 2020.

[8] Songze Li, Salman Avestimehr, et al. Coded computing: Mitigating fundamental bottlenecks in large-scale distributed computing and machine learning. *Foundations and Trends® in Communications and Information Theory*, 17(1):1–148, 2020.

[9] Hodjat Hamidi, Abbas Vafaei, and Seyed Amirhassan Monadjemi. Analysis and design of an abft and parity-checking technique in high performance computing systems. *Journal of Circuits, Systems, and Computers*, 21(03):1250017, 2012.

[10] Andrew I Schein and Lyle H Ungar. Active learning for logistic regression: an evaluation. *Machine Learning*, 68:235–265, 2007.

[11] Tomasz Rymarczyk, Edward Kozłowski, Grzegorz Kłosowski, and Konrad Niderla. Logistic regression for machine learning in process tomography. *Sensors*, 19(15):3400, 2019.

[12] Xiaonan Zou, Yong Hu, Zhewen Tian, and Kaiyuan Shen. Logistic regression model optimization and case analysis. In *2019 IEEE 7th international conference on computer science and network technology (ICCSNT)*, pages 135–139. IEEE, 2019.

[13] Baturalp Buyukates and Sennur Ulukus. Timely distributed computation with stragglers. *IEEE Transactions on Communications*, 68(9):5273–5282, 2020.

[14] James C Bezdek and Richard J Hathaway. Some notes on alternating optimization. In *Advances in Soft Computing—AFSS 2002: 2002 AFSS International Conference on Fuzzy Systems Calcutta, India, February 3–6, 2002 Proceedings*, pages 288–300. Springer, 2002.

[15] Robert L Harrison. Introduction to monte carlo simulation. In *AIP conference proceedings*, volume 1204, page 17. NIH Public Access, 2010.

Appendix A

in this appendix

```
function [error] = MatMin(n,P,k,N,It,LR,seed)
rng(seed)
n_vars = n;
n_nodes = P;
n_threshold = k;

% Create array of alpha and beta
alpha = rand(n_vars,n_nodes);
beta  = rand(n_vars,n_nodes);
% Create decoding coefficients d
number = 1:n_nodes;
combinations = nchoosek(number,n_threshold);
d = rand(n_threshold,size(combinations,1));

% Monte Carlo
% Initialize A and B with random values in the range [0, 1]
Nmc = N; % Number of Monte Carlo
total_A = []; % Create array to store A
total_B = []; % Create array to store B
total_aA = zeros(n_vars,Nmc,n_nodes);
total_bB = zeros(n_vars,Nmc,n_nodes);

% Randomize A and B
% Put them in array
for i = 1:Nmc
    A = (rand(n_vars, 1));
    B = (rand(n_vars, 1));
    total_A = [total_A,A];
    total_B = [total_B,B];
end

% Compute Expected Value
total_AB =  total_A.*total_B;
Expected  = (1 ./ (1 + exp(-sum(total_AB))));
```

```
learning_rate = LR;
max_iterations = It;
Error = [];

% D is the term compute by each node
D = zeros(size(combinations,1),Nmc);

for iteration = 1:max_iterations
    for k = 1:n_nodes
        term(:,k) = 1./(1+exp(-(alpha(:,k)'*total_A).*
        (beta(:,k)'*total_B)));
        expo(:,k) = exp(-(alpha(:,k)'*total_A).*(beta(:,k)'*total_B));
    end

    bB = beta'*total_B;
    AbB = zeros(n_vars,n_nodes,Nmc);
    aA = alpha'*total_A;
    aAB = zeros(n_vars,n_nodes,Nmc);

    for k = 1:Nmc
        for j = 1:n_vars
            for i = 1:n_nodes
                AbB(j,i,k) = -total_A(j,k).*bB(i,k);
                aAB(j,i,k) = -total_B(j,k).*aA(i,k);
            end
        end
    end

    for j=1:size(combinations,1)
        D(j,:) = term(:,combinations(j,:))*d(:,j);
        Di(j,:,1:n_threshold) = term(:,combinations(j,:));
    end

    for j = 1:n_nodes
        for i = 1:size(combinations,1)
            if ismember(j,combinations(i,:))
                setComb(j,i) = 1;
            end
        end
    end

    DD = zeros(size(combinations,1),Nmc);
    for j=1:size(combinations,1)
        DD(j,:) = term(:,combinations(j,:))*d(:,j);
```

```
end
if n_nodes > n_threshold
    for j = 1:size(combinations,1)
        for k = 1:n_nodes
            DF(k,:,j) = setComb(k,j)*DD(k,:);
            ExpectedF(k,:,j) = setComb(k,j)*Expected;

            DForAlpha(k,:,j) = setComb(k,j)*D(k,:);
            ExpectedForAlpha(k,:,j) = setComb(k,j)*Expected;

            DForBeta(k,:,j) = setComb(k,j)*D(k,:);
            ExpectedForBeta(k,:,j) = setComb(k,j)*Expected;
        end
    end
else
    for k = 1:n_nodes
        DF(k,:,1) = setComb(k,1).*DD(1,:);
        ExpectedF(k,:,1) = setComb(k,1)*Expected(1,:);

        DForAlpha(k,:,1) = setComb(k,1).*D(1,:);
        ExpectedForAlpha(k,:,1) = setComb(k,1)*Expected(1,:);

        DForBeta(k,:,1) = setComb(k,1).*D(1,:);
        ExpectedForBeta(k,:,1) = setComb(k,1)*Expected(1,:);

    end
end

for i = 1:size(combinations,1)
    for j = 1:Nmc
        for k = 1:n_nodes
            DExpoA(:,j,i) = DForAlpha(:,j,i)*expo(j,k);
            DExpoB(:,j,i) = DForBeta(:,j,i)*expo(j,k);
        end
    end
end

DDXA = zeros(size(DExpoA));
DDXB = zeros(size(DExpoB));
X = sum(DF-ExpectedF,1);

for i = 1:size(combinations,1)
    DDXA(:,:,i) = DExpoA(:,:,i).*X(:,:,i);
    DDXB(:,:,i) = DExpoB(:,:,i).*X(:,:,i);
end
```

```
    sumDDXA = 1/n_threshold*sum(DDXA,3);
    sumDDXB = 1/n_threshold*sum(DDXB,3);

    Y1 = zeros(size(AbB));
    Y2 = zeros(size(aAB));
    for i = 1:Nmc
        for j = 1:n_nodes
            Y1(:,j,i) = sumDDXA(j,i).*AbB(:,j,i);
            Y2(:,j,i) = sumDDXB(j,i).*aAB(:,j,i);
        end
    end
    Z1 = (1/Nmc)*sum(Y1,3);
    Z2 = (1/Nmc)*sum(Y2,3);

    for i = 1:n_vars
        for j = 1:n_nodes
            gradientA(i,j) = -(2/Nmc)*Z1(i,j);
            gradientB(i,j) = -(2/Nmc)*Z2(i,j);
        end
    end

    for i = 1:size(combinations,1)
        gradientD(1:n_threshold,i) = (2/Nmc)*sum((D(i)-Expected).*
        Di(i,:,1:n_threshold));
    end

    d = d - learning_rate * gradientD;
    alpha = alpha - learning_rate * gradientA;
    beta = beta - learning_rate * gradientB;

    error = (1/size(combinations,1))*((1/Nmc)*sum(sum(X,3)).^2);
    Error = [Error, error];

    if iteration > 2
        if Error(iteration) > Error(iteration-1)
            break
        end
    end
end

plot(Error)
xlabel('iteration')
ylabel('Error')
grid on
```

```
error
end
```