2.5 המאמר

**2.5.1 הרקע**

אלגוריתם ה- SGD נמצא בשימוש נרחב בפתרון של בעיית האופטימיזציה ביישומים רבים כגון למידה עמוקה, בזכות הביצועים המעולים של האלגוריתם במיוחד עם מסדי נתונים גדולים. עם זאת, SGD הוא מטבעו אלגוריתם רציף שבו במקום לחשב את השיפוע במדויק, כל איטרציה מעריכה את השיפוע על בסיס דוגמא יחידה אקראית x, בהתבסס על הנוסחה הבאה:

– הגרדיאנט, בהמשך נגדיר את G בצורה מדויקת

α – שיעור הלמידה

הגרדיאנט הוא פונקציה של x ושל (שניהם וקטורים רב-ממדיים), כאשר רק הינה פרמטר ואילו x למעשה ״נתון״. ניתן לראות בגרדיאנט פונקציה רב-ממדית, כאשר מטרת אלגוריתם ה-SGD היא למצוא את ה- אשר תביא את הפונקציה לערך מינימום (לוקאלי). התהליך מתחיל מנקודה (אקראית) כלשהי וממשיך לנוע במורד היריעה (בכיוון הגרדיאנט- לפי הגרדרת הגרדיאנט) בצעדים בגודל , ובכל איטרציה/צעד בוחרים את ה- אשר יכוונו את הגרדיאנט לשיפוע המרבי באותו צעד.

בנוסחה (i) ניתן לראות כי העדכון בשלב t + 1 תלוי באופן ישיר ב- t הצעדים הקודמים ולכן מגביל את המקביליות של SGD. יתר על כן, וקטור המודל גדול מאוד כך ששיתוף וקטור המודל על פני מספר ליבות על CPU גורם ל"תקשורת" רבה בין הליבות הפנימיות (inter-processor communication ).

כמו כן, החישובים שמתבצעים בכל גישה לזיכרון נמוכים מאוד, אשר יחד עם הדלילות הקיצונית של הנתונים הופכת את מערכת הזיכרון ל"צוואר הבקבוק" הראשי בחישוב SGD על CPUs וגם על GPUs. לכן, היה עניין משמעותי בשנים האחרונות לשפר את הביצועים של SGD על מעבד קונבנציונאלי.

במאמר זה מוצעת סכמה לניצול הזיכרון המוטבע ולהתגברות על "צוואר הבקבוק" של מערכת הזיכרון המגבילה את יכולת הגמישות המקבילית של SGD. הדבר אפשרי ע"י יצירת מערכת זיכרון המותאמת אישית אל מערך הנתונים הספציפי ואל תבניות הגישה המיוחדות לזיכרון בשלבים השונים של האלגוריתם gradient descent. בזכות זה ניתן לצמצם לחלוטין את הצורך בתקשורת פנימית במעבד באלגוריתם descent mini-batch gradient ובכך לשפר את גמישות היישום.

**2.5.2 הגדרת הבעיה ומעבר על הפתרון המתמטי**

כמפורט בהקדמה (ראה סעיף 2.3 ) אלגוריתם SGD מחשב את הגרדיאנט (ואת פונקציית השגיאה) עבור כל דוגמה במערך האימון. קבוצת נתונים המשמשת לחישוב שגיאת המודל מכונה Batch.

מתמטית, עבור Batch בגודל B ועבור הדגימה הפותחת x העדכון של וקטור המודל ניתן על ידי:

J – פונקציית ההפסד התלויה ביישום, כגון למידה מעמיקה

– הגרדיאנט, בהמשך נגדיר את G בצורה מדויקת

α – שיעור הלמידה

θ – וקטור המודל

פונקציה ההיפותזה h

בהנתן דגימת הקלט x , פונקציה ההיפותזה h מסייעת לחזות את ערך המוצא y. עבור משימות רגרסיה, פונקציה ההיפותזה מוגדרת באופן הבא:

עבור רגרסיה לינארית (המקרה הרציף) –

עבור רגרסיה לוגיסטית (המקרה הבדיד) –

פונקציית ההפסד J

לאחר חישוב של פונקציית ההיפותזה h נוכל לחשב את פונקציית ההפסד אשר נתונה ע"י הנוסחה הבאה:

עבור רגרסיה ליניארית (המקרה הרציף) -

עבור רגרסיה לוגיסטית (המקרה הבדיד) -

A – מטריצת הנתונים כאשר כל שורה מייצגת דוגמאות והעמודות מייצגות את התכונות

xj – מדגם התכונות בשורה ה-j במטריצה A

yi – מוצא המערכת בדגימה ה-i

h – פונקציה ההיפותזה המתאימה

השיפוע של פונקציית ההפסד נתון עך ידי:

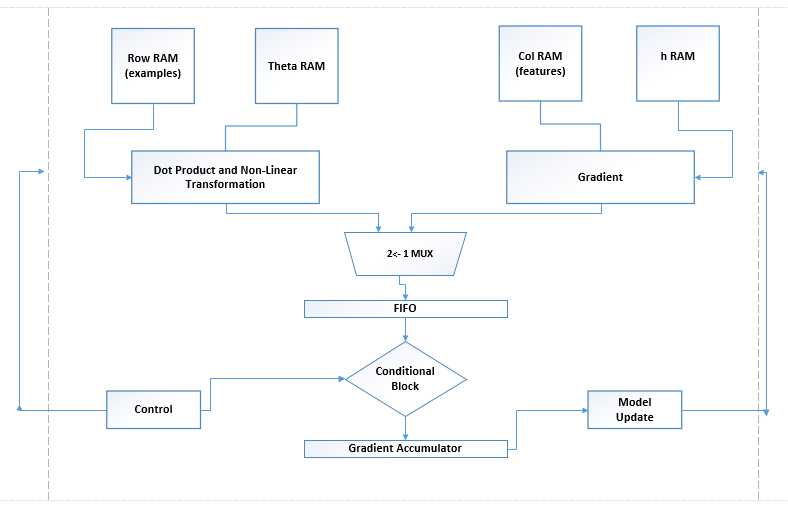
לסיכום, המטרה של האלגוריתם מיני batch המבוסס על gradient decent מקבילי (MBGD) , היא לחשב את השיפוע של דוגמאות batch קטנות, אשר ניתן לחשב במקביל על ידי ליבות שונות. חישוב זה ניתן לחלק לשלושה שלבים כמפורט להלן:

• שלב 1: חישוב המכפלה הפנימית וטרנספורמציה הלא לינארית של:

• שלב 2: חישוב ההגרדיאנט כאשר y הוא פלט צפוי (בלמידה מבוקרת), עבור דוגמה מתאימה של x.

• שלב 3: עדכון המודל

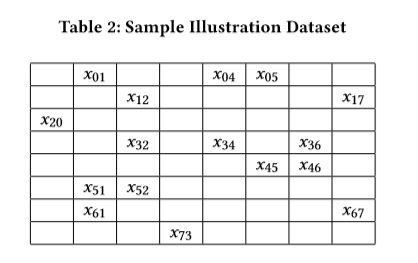
**2.5.3 הארכיטקטורה**

איור 5 מתאר את הארכיטקטורה ב-high level של כל אלמנט עיבוד (PE) המיישמת את שלושת השלבים של האלגוריתם כפי שהם מפורטים לעיל.

איור 5- דיאגרמת בלוקים של הארכיטקטורה עבור אלמנט עיבוד יחיד (single PE )

**2.5.3.1 יחידות הזיכרון**

כל אלמנט עיבוד (PE) מורכב מ- 4 בלוקים של זיכרון. החלוקה מתבססת על אופן הצגת הבעיה במאמר, כלומר הבעיה מתואר כמטריצה דו מימדית כאשר השורות מכילות מידע על קבוצת הדוגמאות (קבוצת המדגם-example) ואילו העמודות מכילות מידע על התכונות של אותן הדוגמאות (features) כפי שניתן לראות באיור 6. במטריצה ישנם תאים ריקים ואילו התאים שאינם ריקים מכילים מידע המסומן ע"י xij .



איור 6 – דוגמא מהמאמר עליו מתבססת הארכיטקטורה המוצעת

להלן פירוט המידע שמכילה כל יחידת זיכרון:

RowRAM- מכיל את השורות של המטריצה ובפרט את האלמנטים שאינם אפס.

colRAM - מכיל את העמודות של המטריצה ובפרט את האלמנטים שאינם אפס.

thetaRAM – מכיל את וקטור המודל θ, המשקלים של ה-MBGD שמתעדכנים בסיום התהליך החישובי באיטרציה הנוכחית.

hRAM – מכיל את תוצאות השלב הראשון, לאחר חישוב פונקצית ההיפוטזה.

\*\*מפורט בהמשך\*\*\*\*

הכתיבה והקריאה מהזיכרון מתבצעת בשיטת indirect access. יוזם הפעולה הוא ה-CPU אך הוא אינו יכול לגשת ישירות לזיכרון לכן התהליך מתבצע בעזרת ה-slave. הממשק בין ה-CPU ל-slave מתבסס על פרוטוקול התקשורת APB כפי שמתואר בחלק 2.4 בהקדמה. ה-CPU שולח בקשת קריאה או כתיבה ל-slave וה-slave "מעביר" זאת לזיכרון (נתאר את התהליך באופן מפורט בהמשך).

**יחידות האחריות על שלב 1 ושלב 2 באלגוריתם MBGD**

ליבת האלמנט החישובי (PE) היא בשני בלוקים מרכזיים:

1. בלוק Dot product and Non Linear Transformation אחראי על חישוב המכפלה הפנימית ופונקציית ה-sigmoid (פונקציה לוגיסטית) עבור שלב 1.
2. בלוק Gradient אחראי על חישוב גרדיאנטים הספציפיים כפי שמפורט בשלב 2 של האלגוריתם.

כעת נפרט על תהליך החישוב בהתאם לדוגמא שמוצגת במאמר שכאמור, מתבססת על סידור הנתונים במטריצה דו מימדית כמתואר באיור 6 לעיל.

בלוק - Dot product and Non Linear Transformation

תהליך החישוב מתחיל בביצוע של המכפלה הפנימית בין המידע שנקרא מזיכרון row RAM למידע ההתחלתי שנקרא מזיכרון theta RAM. במאמר מציעים לממש זאת ע"י הכפלה של כל שורה במטריצה הדו מימדית בוקטור העמודות θ.

נקבל סכום מכפלות מהצורה:

לכל , מחשבים את ה-Z המתאים ומפעילים עליו טרנספורמציה לא ליניארית בצורת פונצקיית ה-sigmoid (מימוש כטבלת LUT) כלומר .

מכיוון שהמטריצה בגודל 8x8, החישוב המוצג המאמר:

בבלוק זה מחושב פונקצית hi יחידה (המימוש המלא מפורט בחלק של ה-implementation). לאחר מכן, תוצאות החישוב מועברות דרך ה-mux ל- FIFO. משם דרך Conditional Block התוצאות מועברת לשמירה בשורה המתאימה בזיכרון ה-h RAM.

מכיוון שהדוגמא מורכבת משמונה שורות, יתבצעו שמונה חישובים דומים ולכן ה-mux ישנה את כיוונו לאחר שמונה מחזורי שעון.

בלוק - Gradient

כאמור, בלוק זה אחראי על מימוש השלב השני באלגוריתם MBGD שמתאר את חישוב הגרדיאנט לפי הנוסחה: .

תהליך החישוב מתחיל בביצוע של המכפלה הפנימית בין המידע שנקרא מזיכרון col RAM למידע שנקרא מזיכרון h RAM (זיכרון h RAM מכיל את תוצאות החישוב של הבלוק ( Dot product and Non Linear Transformation).

החישוב המתואר במאמר:

כפי שניתן לראות, חישוב של gi יחיד מתבסס לעיתים על מספר פונקציות hi שונות לכן נצטרך לקרוא מהזיכרון h RAM מספר פעמים עבור חישוב יחיד (ב-worst case נצטרך לקרוא את כל תאי הזיכרון וזה יקרה כאשר בכל תא במטריצת הנתונים יהיה מידע).

בסיום החישוב של כל יחידת gi , תוצאות החישוב מועברות דרך ה-mux ל- FIFO ומשם דרך Conditional Block התוצאות מועברת לבלוק Gradient Accumulator.

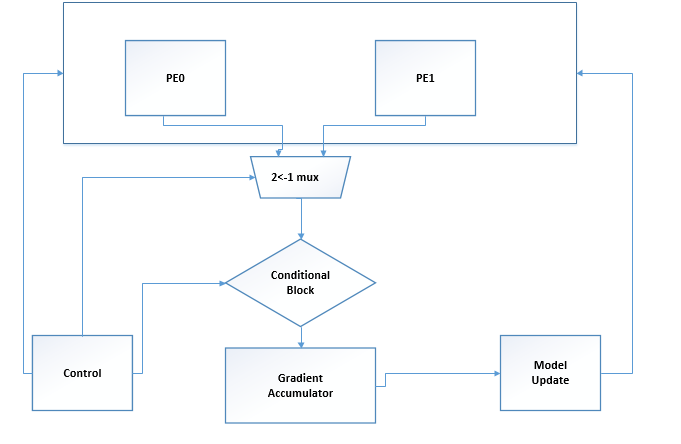
**2.5.3.3 יחידה האחראית על השלב השלישי באלגוריתם MBGD**

הבלוק Model Update אחראי על ביצוע התהליך המתואר בשלב השלישי של האלגוריתם כמתואר בנוסחה: לשם כך, נדרש לקרוא את המידע ההתחלתי ששמור בזיכרון theta RAM, לעשות את החישוב המתאים ולבסוף לעדכן את הזיכרון theta RAM עפ"י תוצאות החישוב.

שאלה למחשבה: גם פה לא כל כך מובן לי אם עושים את זה שורה שורה או הכל ביחד. לדעתי בשביל זה צריך להבין את המימוש של הבלוק Gradient Accumulator. האם כל פעם הוא שומר gi  אחת , מעדכן ואז מכניס עוד אחת וככה ממשיכים?

**2.5.3.4 יחידה האחראית על השלב השלישי באלגוריתם MBGD**

הארכיטקטורה המוצעת יכולה להיות מותאמת למספר אלמנטי חישוב (PE). נציג דוגמא לחישוב המחולק לשני אלמנטים כמתוא באיור 7.



איור 7- דיאגרמת בלוקים של הארכיטקטורה עבור שני PE

במקרה זה, קיים העתק של וקטור המודל θ ווקטור התוצאות של השלב הראשון בזיכרון hRAM בכל אחת מהיחידות. בנוסף לכך, החישוב של השלב הראשון ושני באלגוריתם מתחלק בין אלמנטי החישוב באופן הבא:

שלב 1-

שלב 2-

שלב 3- עדכון המודל במקרה זה מורכב יותר. תחילה מעדכנים את המודל כל בסיס התוצאות שחושבו ב- PE0, זאת אומר ה-mux בדיארמת בלוקים באיור 7, יבחר את הכניסה 0 במשך 4 מחזורי שעון ולאחר מכן המודל יתעדכן על בסיס התוצאות שחושבו ב-PE1.