

**A SOFTWARE PROGRAMMABLE MEMORY ARCHITECTURE
FOR DEEP NEURAL NETWORKS (DNN)**

BY
ALI OMAR ALI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

April 2021

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

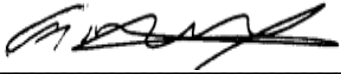
DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

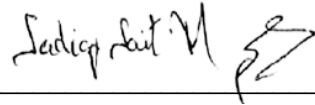
This thesis, written by **ALI OMAR ALI** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING**.



Dr. Muhamed Fawzi Mudawar
(Advisor)



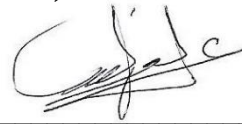
Dr. Aiman El-Maleh (A)
Department Chairman



Dr. Sadiq M. Sait
(Member)



Dr. Suliman Saleh Al-Homidan
Dean of Graduate Studies



Dr. Abdulaziz SalahTabakh
(Member)

Date



© ALI OMAR ALI

2021

To the memory of my Mother,
To my Father,
To my Wife, Brothers, and Sisters

ACKNOWLEDGMENTS

First of all, all thanks and praise go to Allah, who gives me the patience, ambition, and courage to complete my thesis. I sincerely thank my advisor Dr. Muhamed Fawzi Mudawar, for providing directions and support during my research work. This work has not been possible without his help. I am very grateful to Dr. Mudawar. He enriched me with several advanced topics in the Computer architecture domain. Also, Dr. Mudawar devotes some of his precious time to helping me introduce the ISA for M-architecture and the proofreading of my thesis.

I would also like to express my deepest appreciation and gratitude to my committee members Dr. Sadiq M. Sait and Dr. Abdulaziz Tabakh, for their cooperation, help, sparing time to review, proofread and evaluate my work.

I would also like to thank my colleague Mr. Mumin Omar Adam, for his support in developing the simulator.

I would also like to thank the COE department and King Fahd University of Petroleum & Minerals (KFUPM) for allowing me to develop both academically and personally during my life at KFUPM. Also, thanks to my colleagues and friends at KFUPM who made my study and life at KFUPM enjoyable and exciting.

I thank the Kingdom of Saudi Arabia for providing me a fully-funded MSc. scholarship.

Finally, my special appreciations go to my beloved my wife, father, mother, brothers, and sisters for their endless prayers, encouragements, love, and support.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VI
LIST OF TABLES.....	IX
LIST OF FIGURES.....	X
LIST OF ABBREVIATIONS AND NOMENCLATURE.....	XIII
ABSTRACT	XIV
ملخص.....	XVI
CHAPTER 1 INTRODUCTION.....	1
1.1 Problem Statement	4
1.2 Thesis Objectives and Contributions	5
1.3 Methodology	6
1.4 Thesis Organization	7
CHAPTER 2 BACKGROUND	8
2.1 Background of Artificial Intelligence.....	8
2.1.1 Neural Networks (NNs).....	12
2.1.2 Deep Neural Networks (DNNs).....	14
2.1.3 Algorithms for the Neural Networks	15
CHAPTER 3 LITERATURE SURVEY.....	36
3.1 Overview of Machine Learning Architecture	36
3.2 Neural Networks Architectures.....	40
3.2.1 LeNet	40
3.2.2 AlexNet	41
3.2.3 ZfNet	41

3.2.4	VGGNet.....	42
3.2.5	GoogLeNet.....	43
3.3	Neural Networks Instruction Set.....	43
CHAPTER 4 THE PROPOSED INSTRUCTION SET ARCHITECTURE		49
4.1	Overview of the M-Architecture ISA	49
4.2	General-Purpose Registers (GPRs)	50
4.3	Program Counter Register	52
4.4	Instruction Set Formats	53
4.5	Types of Instructions	54
4.5.1	Scalar ALU Instructions	55
4.5.2	Scalar FPU Instructions	56
4.5.3	Control Flow Instructions	57
4.5.4	Load and Store Memory Instructions	59
4.5.5	Read and Write Instructions	60
4.5.6	Integer Memory Computation Instructions	66
4.5.7	Floating-Point Memory.....	68
4.5.8	Memory Reductions Instructions	70
4.6	Code example for some of our machine learning operations	71
4.6.1	Dot Product.....	71
4.6.2	Vector Matrix Multiplication in a Neural Network	73
CHAPTER 5 THE PROPOSED ARCHITECTURE		78
5.1	M-architecture Overview.....	78
5.2	Software Programmed Local Memory	81
CHAPTER 6 EXPERIMENT DESIGN AND SIMULATION		86
6.1	Experiment Design.....	86
6.1.1	Convolutional Neural Networks (CNNs) Algorithm Implementation.....	86
6.1.2	Recurrent Neural Networks (RNNs) Implementation	96
6.2	Simulation of the Experiment	98
6.3	Verification of the M-Simulator	101
CHAPTER 7 BENCHMARK RESULTS		109

7.1	Convolution Neural Networks	109
7.1.1	Convolution Operation	109
7.1.2	Pooling Operation.....	116
7.1.3	Fully-Connected Layer	117
7.1.4	SoftMax Operation	120
7.2	Recurrent Neural Networks	125
7.3	Performance Evaluation	126
7.3.1	Results and Discussions.....	127
CHAPTER 8 CONCLUSION AND FUTURE WORK		137
REFERENCES.....		139
VITAE		146

LIST OF TABLES

Table 2. 1 list of the common activation functions for Artificial Intelligent, AI [17] [26] [27].	11
Table 2. 2 Histories of the neural networks	13
Table 2. 3 Common algorithms in machine learning [46] [47].	16
Table 2. 4 History of the CNN Models.....	18
Table 2. 5 Sample of 1-Hot Representation.....	33
Table 3. 1 An Overview to Cambricon Instructions [94]	44
Table 6. 1 The CNN's Output Layer Sizes after Operation	88
Table 6. 2 Weight Initialization Parameters	92
Table 6. 3 Data Type and Size of the Training Parameters	95
Table 6. 4 The Selected Sentences in 1-Hot Representation	97
Table 6. 5 Performance Equations Mode in M-Simulator.....	99
Table 6. 6 Configurable Parameters for our Experiment.....	100
Table 7. 1 Number of Operation Per Cycles in MNIST Benchmark	127
Table 7. 2 Number of Operation Per Cycles in CIFAR-10 Benchmark	129
Table 7. 3 Training Output for the Labels in CIFAR-10.....	131
Table 7. 4 Training Output for the Labels in MNIST-10	132
Table 7. 5 Number of Operation Per Cycles in RNNs.....	133
Table 7. 6 Number of clock cycles of the operation implemented by different instructions.....	135

LIST OF FIGURES

Figure 1. 1 Artificial Neuron Networks Parameters	2
Figure 1. 2 Artificial Neuron Networks Layers	2
Figure 2. 1 Relationship of the Artificial Intelligence fields	9
Figure 2. 2 Natural Neuron Networks [25]	10
Figure 2. 3 Artificial Neuron Networks	10
Figure 2. 4 The plot of the Speed Performance against Amount of Data [44] [45].....	14
Figure 2. 5 Type of Learning in Artificial Neural Network	15
Figure 2. 6 Convolutional Neural Network (CNN) Algorithms	17
Figure 2. 7 MNIST Benchmark [61].....	20
Figure 2. 8 CIFAR-10 Benchmark [62].....	20
Figure 2. 9 ImageNet Dataset Benchmark	21
Figure 2. 10 Demonstration of Convolution Computation in CNN for One Channel	22
Figure 2. 11 Demonstration of Convolution Computation in CNN for Three Channels.....	22
Figure 2. 12 The cracking of Convolution and Deconvolution Operation	23
Figure 2. 13 Demonstration of Pooling Computation.....	24
Figure 2. 14 Flattening Operation	24
Figure 2. 15 Sample of Fully-Connected Layer.....	25
Figure 2. 16 Fully Connected Layer and Softmax	27
Figure 2. 17 Fold and Unfold Structure of the RNN Model [66]	28
Figure 2. 18 RNN operations and Parameters [66].....	29
Figure 2. 19 Categories of RNN Model [66]	30
Figure 2. 20 Long Short Time Memory Dependencies Block Diagram [73]	31
Figure 2. 21 Gated Recurrent Unit (GRU) [79].....	32
Figure 2. 22 Sample of Word Embedding	34
Figure 2. 23 Angle of Similarity between the words [66]	35
Figure 3. 1 LeNet Architecture [90]	40
Figure 3. 2 AlexNet Architecture [45].....	41
Figure 3. 3 ZfNet Architecture [44].....	42
Figure 3. 4 VGGNet Architecture [44]	42
Figure 3. 5 GoogleNet Architecture [29].....	43
Figure 3. 6 Matrix Multi-Vector (MMV) Instruction [92]	45
Figure 3. 7 Vector Greater Than Merge (VGTM) instruction [92]	45
Figure 3. 8 Cambricon program fragments of MLP, pooling, and BM [92]	45
Figure 3. 9 RISC-V architecture instruction space [93]	46
Figure 3. 10 E203 coprocessor extended instruction encoding format [93]	46
Figure 3. 11 Instruction types and their formats for the designed ANN processor [94].....	47
Figure 3. 12 An example RNN program using our custom-developed ISA [94]	47
Figure 4. 1 Software Programmed Local Memory	49
Figure 4. 2 Length Registers.....	50

Figure 4. 3 General-Purpose registers.....	51
Figure 4. 4 Register Usage and the Function Call Standard	52
Figure 4. 5 Program Counter Register	52
Figure 4. 6 Instruction Set Formats.....	53
Figure 4. 7 Instruction Formats For Scalar ALU	56
Figure 4. 8 Format for Scalar FPU Instruction	56
Figure 4. 9 Branch Instruction Format.....	57
Figure 4. 10 Unconditional Branch Instruction Format.....	58
Figure 4. 11 Jump-and-Link Register Instruction Format	58
Figure 4. 12 Load Instruction Format	59
Figure 4. 13 Instructions for LM Opcode	60
Figure 4. 14 Instructions for SM Opcode	60
Figure 4. 15 The Read and Write Instructions Format	61
Figure 4. 16 Push Instruction Format	64
Figure 4. 17 Pop Instruction Format	65
Figure 4. 18 Formats for Integer Memory Computation Instructions	67
Figure 4. 19 Format for Floating-Point Memory Computation Instructions	68
Figure 4. 20 Memory Reduction Instruction Format	70
Figure 4. 21 Vector-Matrix Multiplication	73
Figure 5. 1 High-Level Block Diagram of the M-Processor	80
Figure 5. 2 Mapping a Local byte Address, A onto a Slice, S, and Index I.....	81
Figure 5. 3 Software Programmed Memory Diagram For Four Slices.....	82
Figure 5. 4 Software Programmed Memory Diagram For 64 Slices	83
Figure 5. 5 Connecting four slices to a leaf router.....	85
Figure 6. 1 The proposed Structure for CNN Algorithms	87
Figure 6. 2 MNIST Benchmark Properties [61]	89
Figure 6. 3 Python Codes for MNIST Matrix Extraction	90
Figure 6. 4 Python Codes for CIFAR-10 Matrix Extraction	90
Figure 6. 5 Sample of the Matrix Weights values are Generated	93
Figure 6. 6 Convolution Simplification by using Toeplitz Matrix for the Kernel Matrix	93
Figure 6. 7 Keras Kernel Matrixes [100]	94
Figure 6. 8 Python code to Generate Toeplitz Matrix for the Filter	94
Figure 6. 9 Implementation Design of the RNNs Diagram for the Three Sentences	96
Figure 6. 10 Python script for Generate 1-Hot Word Embedding.....	97
Figure 6. 11 Python script for Generate Vector in Word2Vec Method.....	98
Figure 6. 12 M-Simulator User Interface (UI).....	99
Figure 6. 13 Sample of the Configurable Parameters in M-Simulator	100
Figure 6. 14 Sample of the Running Experiment (Convolution).....	101
Figure 6. 15 Verification Input Matrix (X) and Filter Matrix (K)	102
Figure 6. 16 Sample of the Convolution Operation in Python	102

Figure 6. 17 Input Vector (X) and Kernel Toeliptz Matrix (K) for M-Simulator	103
Figure 6. 18 Loaded of the Input X and Kernel Matrix Byte Elements in Global Memory	103
Figure 6. 19 Sample of the Convolution Operation in M-Simulator	104
Figure 6. 20 Result of the Convolution Operation in M-Simulator	105
Figure 6. 21 Sample of Maximum Pooling Operation in Python	107
Figure 6. 22 Maximum Pooling Operation in M-Simulator	107
Figure 6. 23 The Output Result of Maximum Pooling Operation in M-Simulator	108
Figure 7. 1 Convolution Implementation in SPM	110
Figure 7. 2 Matrix-Matrix Multiplication for Convolution Implementation	111
Figure 7. 3 The Convolution Matrixes in CIFAR-10 when $c > \text{MRL in K}$	114
Figure 7. 4 Example of Maximum Pooling Operation	116
Figure 7. 5 Fully-Connected Layer Connection	117
Figure 7. 6 M-Architecture Implementation of forwarding Propagation	118
Figure 7. 7 Updating Weights Analysis	122
Figure 7. 8 Updating Weights Implementation	122
Figure 7. 9 MNIST Benchmark CNNs (Operations Per Cycles)	128
Figure 7. 10 MNIST SpeedUp Normalize to Single Execution Unit	129
Figure 7. 11 Number of Operations Per Cycles in CIFAR-10 Benchmark	130
Figure 7. 12 SpeedUp Normalize to Single Execution Unit (CIFAR-10 Benchmark)	130
Figure 7. 13 The Obtained Estimated Label Values of CIFAR-10 Benchmark	132
Figure 7. 14 The Obtained Estimated Label Values of CIFAR-10 Benchmark	133
Figure 7. 15 RNN Encoding Methods (Number of Operations Per Cycle)	134
Figure 7. 16 RNN Encoding Methods (SpeedUp)	135
Figure 7. 17 M-Architecture Acceleration Ratio of the CNN Operations	136

LIST OF ABBREVIATIONS AND NOMENCLATURE

AI	:	Artificial Intelligence
ANN	:	Artificial Neural Networks
CIFAR-10	:	Canadian Institute For Advanced Research-10 images
CNN	:	Convolution Neural Networks
CPU	:	Central Processing Unit
DNN:	:	Deep Neural Networks
GRU	:	Gated Recurrent Unit
LSTM	:	Long Short-Term Memory
MNIST	:	Modified National Institute of Standards and Technology
NEU	:	Number of Execution units
RNN	:	Recurrent Neural Networks

On-chip memory is meant to keep SRAM memory on the processor chip. in this thesis, either cache or SPM is called on-chip memory, while the main memory is referred to as off-chip memory.

ABSTRACT

Full Name : ALI OMAR ALI
Thesis Title : A SOFTWARE PROGRAMMABLE MEMORY (SPM)
ARCHITECTURE FOR DEEP NEURAL NETWORKS (DNN)
Major Field : Computer Engineering
Date of Degree : Apri 2021

Deep Neural Networks (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. DNN has many applications such as computer vision, language processing, drug discovery and toxicology, medical analysis, financial fraud detection, and military operations.

However, due to the complex structure involving many operations and large amounts of data, the implementation of DNN algorithms requires powerful machines that can support the workload. Moreover, general-purpose processors are not well suited for implementing DNN algorithms in terms of performance. Several architectures (e.g., LeNet, GoogleNet, VGGNet, ZfNet, AlexNet) have been proposed to overcome the difficulty of DNN acceleration in hardware, but are not yet flexible enough to be a standard. The DNN algorithms such as Convolution Neural Networks (CNN) and Recurrent Neural Networks (RNN) are among the popular algorithms used in computer vision and natural language processing applications.

In this research work, we have proposed an efficient and scalable Software Programmable Memory (SPM) architecture for design data parallelism to support CNN and RNN with better performance. We have also proposed an efficient and flexible instruction set to resolve complex functional blocks of CNN and RNN operations into general-purpose instructions corresponding to low-level computational operations such as dot product, vector-matrix multiplication, and matrix-matrix multiplication. The performance in the M-architecture is achieved by executing the computations on the SPM which is divided into regions M0, M1,...M15 where each region can accommodate a large vector and a vector execution unit (ExU) which helps to process data in parallel and in less time.

The proposed algorithms were simulated using M-simulator which checks the functionality and performance such as number of operations, number of cycles, and number of operations per cycle. The MNIST and CIFAR-10 benchmarks were implemented their class images in parallel. The Word2vec and 1-Hot representations were also implemented for the basic RNN algorithm. In this work, the DNN algorithms were implemented with the best performance offered by Software Programmable Memory in the M- architecture.

ملخص

الاسم الكامل: علي عمر علي

عنوان الرسالة: بنية الذاكرة القابلة للبرمجة للشبكات العصبية العميقة

التخصص: هندسة الحاسوب

تاريخ الدرجة العلمية: نيسان, 2021

الشبكات العصبية العميقة (DNN) هي شبكة عصبية اصطناعية (ANN) ذات طبقات متعددة بين طبقات الإدخال والإخراج. DNN لديها العديد من التطبيقات مثل رؤية الكمبيوتر ومعالجة اللغة واكتشاف العقاقير وعلم السموم والتحليل الطبي واكتشاف الاحتيال المالي والعمليات العسكرية. ومع ذلك ، نظرًا للهيكल المعقد الذي يتضمن العديد من العمليات وكميات كبيرة من البيانات ، فإن تنفيذ خوارزميات DNN يتطلب آلات قوية بشكل خاص يمكنها دعم عبء العمل. علاوة على ذلك ، فإن المعالجات ذات الأغراض العامة ليست مناسبة تمامًا لتنفيذ خوارزميات DNN من حيث الأداء. لقد تم اقتراح العديد من البنى (على سبيل المثال ، LeNet و GoogleNet و VGGNet و ZfNet و AlexNet) للتغلب على صعوبة تسريع DNN في الأجهزة ، ولكنها ليست مرنة بعد بما يكفي لتكون معيارًا. تعد خوارزميات DNN مثل الشبكات العصبية الالتفافية (CNN) والشبكات العصبية المتكررة (RNN) من بين الخوارزميات الشائعة المستخدمة في رؤية الكمبيوتر وتطبيقات معالجة اللغة الطبيعية.

في هذا العمل البحثي ، اقترحنا بنية ذاكرة برمجية قابلة للبرمجة (SPM) فعالة وقابلة للتطوير لتوازي بيانات التصميم لدعم CNN و RNN بأداء أفضل. لقد اقترحنا أيضًا مجموعة تعليمات فعالة ومرنة لحل الكتل الوظيفية المعقدة لعمليات CNN و RNN في تعليمات للأغراض العامة تتوافق مع العمليات الحسابية منخفضة المستوى مثل المنتج النقطي وضرب المصفوفة المتجهية وضرب المصفوفة. يتم تحقيق الأداء في الهندسة المعمارية M من خلال تنفيذ العمليات الحسابية على SPM والتي تنقسم إلى مناطق M0 و M1 و

... M15 حيث يمكن لكل منطقة استيعاب متجه كبير ووحدة تنفيذ متجه (ExU) مما يساعد على معالجة البيانات بالتوازي وفي وقت أقل. تمت محاكاة الخوارزميات المقترحة باستخدام M-simulator الذي يتحقق من الوظائف والأداء مثل عدد العمليات وعدد الدورات وعدد العمليات لكل دورة. تم تطبيق معايير MNIST و CIFAR-10 على صورهم الطبقة بالتوازي. تم أيضًا تنفيذ تمثيلات Word2vec و Hot-1 لخوارزمية RNN الأساسية. في هذا العمل ، تم تنفيذ خوارزميات DNN بأفضل أداء قدمته ذاكرة برمجية قابلة للبرمجة في بنية M.

CHAPTER 1

INTRODUCTION

Deep Neural Network (DNN) is the artificial neural network (ANN) in the field of machine learning with multiple layers between the input and output layers. The DNN enables a machine to learn, think, process, and develop data abstractions with high performance like the human brain. In 1967, the first machine learning algorithms were published by Lapa and Alexey Ivakhneko called multilayer perceptrons, feedforward, supervised, and deep [1]. The deep algorithm was improved in 1971 by the group method of Data Handling to implement eight trained layers used in their architecture for recognition and computer vision which led to the algorithm being modified again in 1980 by Kunihiko Fukushima [2] [3]. The machine learning community received the official term deep learning in 1986 after it is introduced by Rina Dechter while Artificial Neural Networks accepted the term deep learning in 2000 after it is introduced by Igor Aizenberg and colleagues [4][5][6].

Machine learning becomes an interesting field for our modern technology in terms of research and implementation due to its powerful features in many applications such as computer vision, and natural language processing. Most machine learning techniques are implemented through a special class known as Deep Learning or Deep Neural Networks (DNN) [7][8][9]. The DNN is used in language processing, computer vision, drug discovery and toxicology, bioinformatics and medical analysis, financial fraud detection, and military operations [10][11][12][13][14]. Deep neural networks are a sequence of layers consisting of the input layer of artificial neurons, hidden layer(s),

and output layer. The result of output and hidden layer is obtained after activating the summation of multiplication of the inputs of nonlinear functions and weights as shown in Figure 1.1 [15].

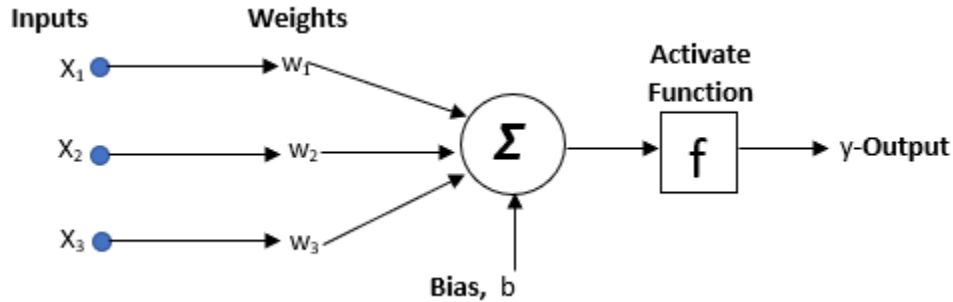


Figure 1. 1 Artificial Neuron Networks Parameters

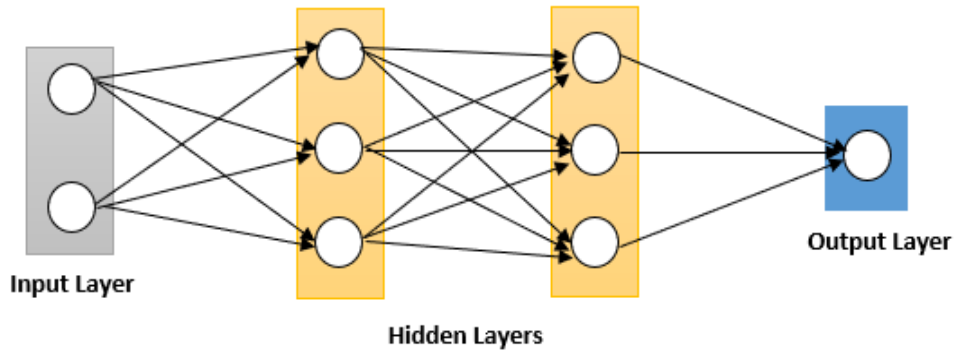


Figure 1. 2 Artificial Neuron Networks Layers

Figure 1.1 shows an artificial neural network, where the input nodes are neurons connected to their weight function by lines called edges to carry the information up to the activation function, the function sums the inputs to produce results. Figure 1.2 explains the structure of the artificial neural network in terms of layers (i.e., input layer, hidden layer, and output layer), the gray color layer is the input data layer, followed by the light yellow layers as a hidden layer between the input and output layer, and the blue layer is the output layer. When the number of hidden layers is very large, the structure is called deep neural networks. It also works by feeding the input data into a forward direction to improve the reconstruction ability of each layer [4][15].

Deep learning algorithms require high-end machines with lots of hardware support instead of existing traditional machines to perform many operations in matrix multiplication and nonlinear functions such as ReLU, hyperbolic tangent, sigmoid, average, and max-pool. For this challenge, deep neural networks became famous after the presence of powerful computers, new technologies, and the availability of a huge dataset since 2011 [16]. The new machines work efficiently for big data by using general-purpose graphics processing units (GPUs), rectifier linear units (ReLU), and Restricted Boltzmann Machine (RBM) under a deep belief network (DBN) [17]. For example, one of the most popular DNN algorithms is Convolutional Neural Network (CNN/ConvNet). ConvNet has led some companies such as Samsung, Intel, NVIDIA, Qualcomm, and Mobileye to develop special ConvNet chips that help them identify services and products based on images. The ConvNet chips provide better performance for real-time vision systems in smart devices such as robots, self-driving cars, smartphones, and cameras [15][18].

Due to the improvement in technology, there are many proposed hardware accelerators to speed up the neural network in the machine. For example, Qualcomm and Apple process neural networks in a special System on the Chips (SoCs), while Google uses Tensor Processing Unit (TPU) dominated by GPUs [19][20]. The structure of most proposed accelerators supports the small subset, same data locality, and computation pattern otherwise it is not flexibly efficient when out of these characteristics. For example, the Boltzmann Machines (BMs) have fully connected neurons, but do not support being implemented in the DaDianNao accelerators, which efficiently implement the Multi-Layer Perceptrons (MLPs) [21][22][23].

Convolution Neural Network (CNN) and Recurrent Neural Network (RNN) are among the deep neural network algorithms that can perform identification of two-dimensional information due to their multilayer perceptron structure. In this research work, we investigate the implementation of

convolutional neural networks (CNN) and RNN in M-architecture using Software Programmable Memory (SPM) architecture which is capable of accelerating the computations and overall performance.

1.1 Problem Statement

Nowadays, software platforms are the most commonly used tools for practical implementations of DNNs, as they can support the learning algorithms in a short time and flexibly handle any change that needs to be updated. However, it is challenging to meet the performance of deep neural network applications such as image processing systems and predictive control, which require very high data throughput.

In terms of hardware, a general-purpose architecture (i.e., using only CPUs) may not be promising enough to implement deep neural networks in terms of energy efficiency and overall performance, since DNNs require many hardware resources to support nonlinear functions with high performance [24]. As we reported in the literature review, many publications and industries such as Google, Qualcomm, IBM, Apple, Facebook, Microsoft, Yahoo, Twitter, and Intel have proposed a hardware accelerator as a solution. Field-Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) are among the proposed accelerator to accelerate the learning algorithms in hardware. However, the proposed hardware accelerators use cache memory instead of scratchpad memory, which is inefficient in terms of predictability and power consumption, etc. Cache memory is not a promising choice to on-chip memory for neural network processing, which inherently has a large amount of data for which a machine needs fast access to achieve better throughput. As reported in the literature review, the machines with Software Programmable Memory (SPM) can provide much speed and high bandwidth to support the workload as they are well suited for implementing learning algorithms.

The main objective of this proposal is to use SPM as a solution for learning algorithms in implementing deep neural networks with high flexibility and high configurability to improve the overall performance and energy efficiency.

1.2 Thesis Objectives and Contributions

As mentioned in the literature review, much work has been done on hardware for learning algorithms, with most of the proposed architectures using cache for general purpose and special small scratchpad memory for an accelerator. SPM allocation techniques have been proposed for accelerating learning algorithms as a solution, but they have not received much attention to improve overall performance. This work explores the advantages of M-architecture design for deep neural networks due to the features of our architecture that supports SPM multi-region data allocation implemented in one dimension (1D). This feature helps in supporting multi-layers required in deep neural networks, and in processing large amounts of data. A new flexible instruction set has been developed to provide the ability to access data in parallel during computation.

The following are the specific objectives of this work:

- To use the SPM in the architecture for a design data parallelism to support Deep Neural Networks (DNN) with better performance
- To propose an efficient and flexible instruction set for DNN learning
- Conducting a performance analysis of the proposed DNN's architecture
- Sharing our findings with the research community

1.3 Methodology

We have followed the below research methodology to conduct this research:

- Extended study and literature survey for DNN in SPM

In this phase, we extensively studied artificial neural networks especially the DNN. We identified the strength and limitations of DNN on how can it benefit within Software Programmable Memory (SPM) to perform better performance.

- Proposing M-architecture

In this phase, we proposed a new architecture that uses on-chip SPM memory to manage the computation of a deep neural network.

- M-architecture ISA

Here, we introduced a new flexible ISA for DNNs in our architecture to simplify computations by allows these instructions to inherit data-level parallelism and support non-linear operations.

- Simulation of the proposed architecture

During this phase, a simple simulator was developed to simulate the SPM architecture ISA for the proposed DNN's algorithms and evaluate its performance.

- DNN Algorithms Evaluation in SPM Architecture

During this phase, the proposed extension of the architecture will be experimented with using a set of benchmarks, from the collected experimental results, the performance will be analyzed and compared to the existing works.

- Concluding and suggesting some future trends

In the end, conclusions from the research work were drawn, and future work directions were advised.

- Thesis writing

The thesis writing process was done during all the mentioned phases.

1.4 Thesis Organization

The remaining parts of this thesis are organized as follows. Chapter 2 gives a detailed background on deep neural networks. The main points of previous related works and their advantages and limitations are discussed in Chapter 3. Chapter 4 gives an overview of the ISA design of the proposed memory architecture, its instruction set and highlights the advantages of the proposed architecture. The presentation and development of the M architecture is presented in Chapter 5. Chapter 6 discusses the experiment design and simulation used to evaluate the performance of the proposed architecture. Chapter 7 presents the benchmarks and performance of the proposed architecture. The summary of this thesis and some suggestions for future work are presented in Chapter 8.

CHAPTER 2

BACKGROUND

This chapter presents the background information about DNNs on how it works, some of its concepts, and its position in artificial intelligence (AI). We will mention the different algorithms used nowadays and previous history according to various applications to motivate the work. Then CNN and RNNs algorithms described in more detail include their structures and tools.

2.1 Background of Artificial Intelligence

Artificial Intelligence is the branch of science and engineering that enables a machine to be intelligent, process queries without being explicitly programmed, and be able to achieve goals like a human. The field of creating intelligent machines, in general, is known as machine learning, which was defined by Arthur Samuel in 1959 [17]. Machine learning consists of brain-inspired computation, which consists of different subdomains known as Neural Networks (NN), Deep Learning, and Spiking, and their relationship can be seen in Figure 2.1. A brain-inspired computation is an algorithm or program that transfers some functions from the way the brain works to the machines. Spiking computing is the subfield within brain-inspired computing in the field of machine learning, which originated from the reason that the communication between axons and dendrites are spike-like impulses that depend on the arrival time of the impulse. The second fact is that the information that is transmitted does not depend on the amplitude of a single spike, but on the computation within the activation function that involves the pulse width and the temporal relationship between different pulses. The spiking part is among the important parts in the computation in the brain, but many results of scientists show that the neuron is the most important

computational element of the brain, and it was found that in the human brain on average about 86 billion neurons are connected with some of them known as dendrites and an axon.

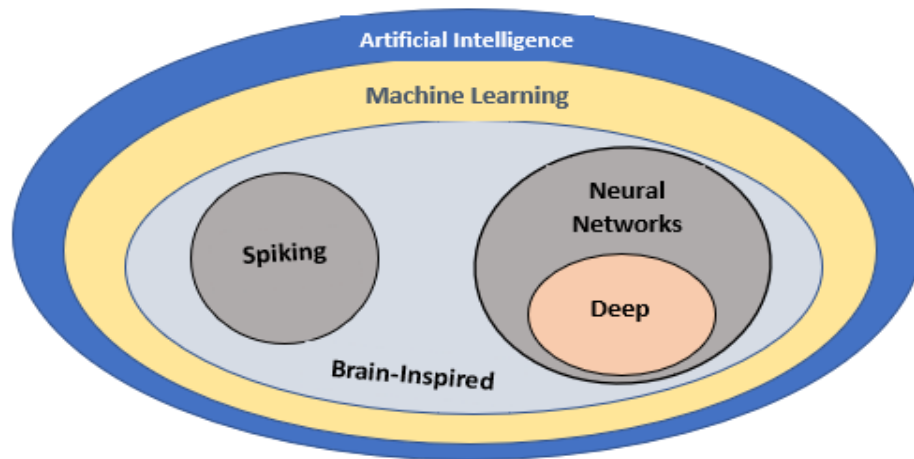


Figure 2. 1 Relationship of the Artificial Intelligence fields

Those neurons which allow the entry of a signal are called dendrites, and those which allow the exit of a signal are called axons. With these properties, the main function of the neuron is to transmit and compute the signal within the brain by entering through dendrites and exiting through an axon, as shown in Figures 2.2 and 2.3. These dendrites and an axon signal are called activations, which are the input and output signals. The next feature of the brain is the synapse, which provides the connection between dendrites and an axon within the networks. The number of synapses can be approximately about 10^{14} to 10^{15} in the human brain [23].

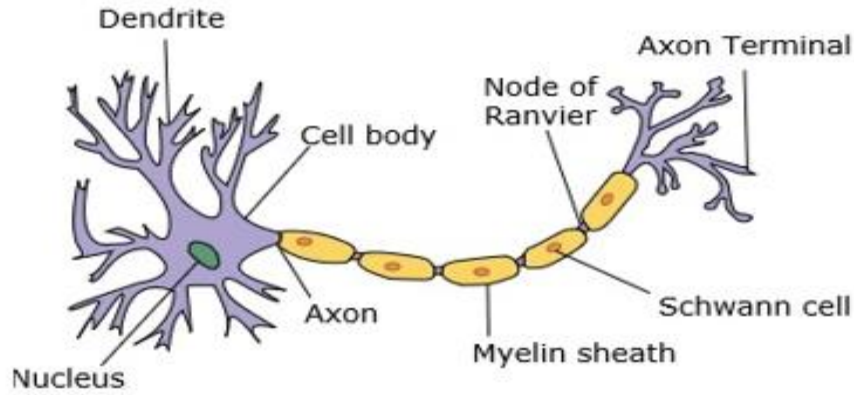


Figure 2. 2 Natural Neuron Networks [25]

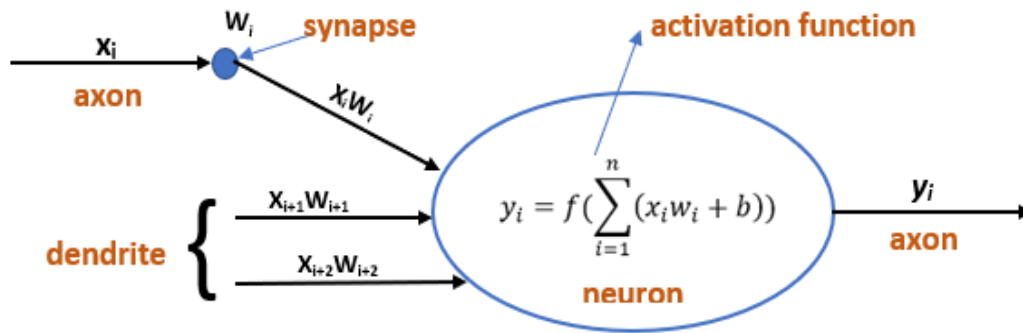
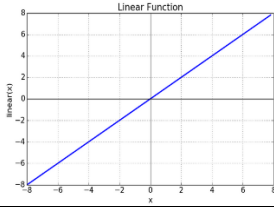
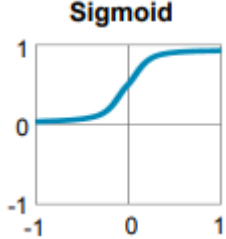
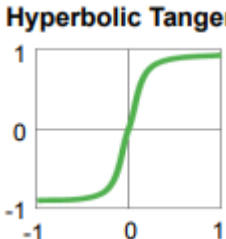
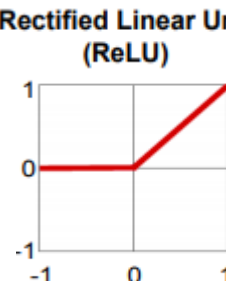
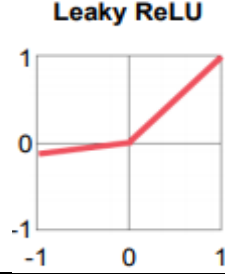
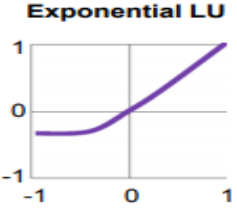


Figure 2. 3 Artificial Neuron Networks

Figure 2.3 explains the structure of artificial neuron networks, which is almost a copy of the natural neuron networks in the human brain. Neurons are the input nodes connected to their weight function by lines called edges to transmit the information until the activation function can sum the inputs to produce results. The synapse plays an important role in artificial neural networks by scaling the signal that passes through the network to enable learning processes. The learning process depends on the changes in the weights associated with the synapses. Therefore, a synapse is the scaling factor and can be referred to as a weight (w_i). Table 2.1 shows the different types of activation functions.

Table 2. 1 list of the common activation functions for Artificial Intelligent, AI [17][26][27]

S/N	TYPE	GRAPHY	FORMULA
1.	Linear		$y = x$
2.	Traditional Non-Linear Activation Functions [28]		$y=1/(1+e^{-x})$
3.			$y=(e^x-e^{-x})/(e^x+e^{-x})$
4.	Modern Non-Linear Activation Functions [29][30].		$y=\max(0, x)$

5.		 Leaky ReLU	$y = \max(\alpha x, x)$
6.		 Exponential LU	$y = \begin{cases} x, & x \geq \theta \\ \alpha(e^x - 1), & x < \theta \end{cases}$

2.1.1 Neural Networks (NNs)

Neural networks are a set of algorithms that are modelled to recognize the relationship between the set of processing data that mimic the type of operations in the human brain. Neural networks can adapt to the changing inputs to produce suitable possible results without having to redesign their criteria. With this fact, neural networks refer to systems of neuron computation that can be either artificial in nature or organic. It controls the relationship between the input and the output by considering the sum of the product of weight and associated input data. Equation 2.1 explains the general relationship between neuron values in the system including an activation function, which can be either a linear or nonlinear function [30].

$$y_i = f\left(\sum_{i=0}^n x_i w_i + b\right) \quad (2.1)$$

Where: w_i is weight value, x_i is an Input activation, y_i is output activation, f is the activation function, and b is the bias term.

The linear activation function causes the output of the neural network to be generated only when the input value changes with the associated weight and its sum is taken. Moreover, it seems to be a nonlinear function when output is generated only when the inputs have reached or exceeded the required threshold value. Artificial neuron models proposed by computational neurobiology have conducted many significant researches. Table 2.2 shows the brief history of neural network evaluation from 1940 to 2014 [17][25].

Table 2. 2 Histories of the neural networks

YEAR	EVALUATION
1940 [31]	Neural networks were proposed
1943 [31]	McCulloch & Pitts show that neurons can be combined to construct a Turing machine
1958 [32]	Rosenblatt shows that perceptrons will converge if what they are trying to learn can be represented
1960 [17]	Deep neural networks were proposed
1969 [33]	Minsky & Papert show the limitations of perceptron's, killing research in neural networks for a decade
1985 [34]	The backpropagation algorithm by Geoffrey Hinton et al. revitalizes the field.
1988 [35]	Neocognitron: a hierarchical neural network capable of visual pattern recognition
1989 [36]	Neural networks for recognizing digits (LeNet)
1998 [37]	CNNs with Backpropagation for document analysis by Yan LeCun
1990 [38]	Hardware for shallow neural nets (Intel ETANN)
2006 [39]	The Hinton lab solves the training problem for DNNs.
2011 [40]	Breakthrough DNN-based speech recognition (Microsoft)
2012 [41]	DNNs for vision start supplanting hand-crafted approaches (AlexNet)
2014 [42]	Rise of DNN accelerator research (Neuflow, DianNao...)

2.1.2 Deep Neural Networks (DNNs)

Deep neural networks are the domain of neural networks that have more than one hidden layer between input and output layers. Deep neural networks allow a machine to retrieve raw data and automatically analyze the representation, either for classification or recognition. Deep neural networks provide a maximum score for the correct answer and a minimum score for the incorrect answer during training, unlike traditional machine learning, as shown in Figure 2.4.

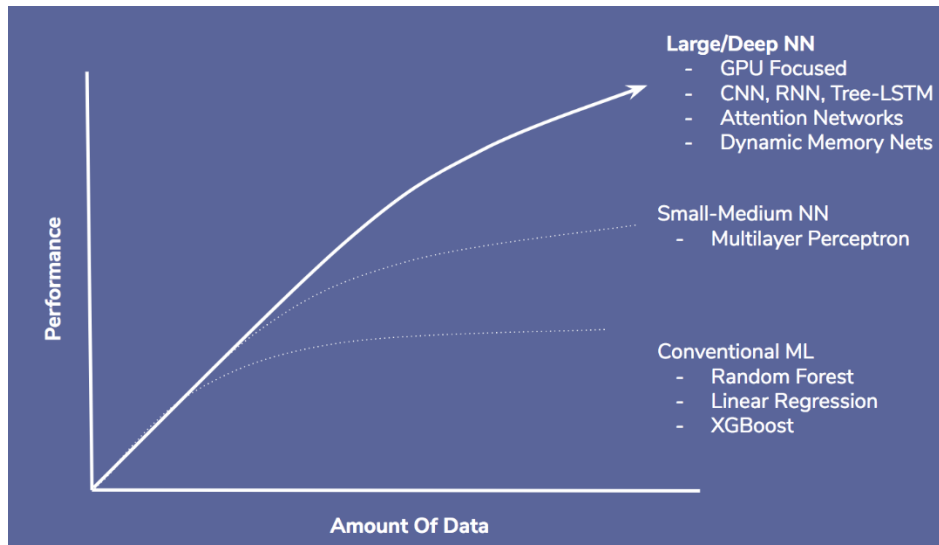


Figure 2. 4 The plot of the Speed Performance against Amount of Data [43][44]

In training a network, the gradient descent optimization process is usually used to update the weight (W_{ij}) in the network by determining the weight loss through partial derivative for each weight as follows:

$$W_{ij}^{t+1} = W_{ij}^t - \alpha \frac{\partial L}{\partial W_{ij}} \quad (2.2)$$

Whereby: α is the learning rate, W_{ij}^t is the training weight at the current time (t), W_{ij}^{t+1} is the updated training weight. $\frac{\partial L}{\partial W_{ij}}$ is the derivative of the error with respect to weight (W)

2.1.3 Algorithms for the Neural Networks

Neural Network Algorithms are those programs that change by learning to perform their given tasks, and this can be done either by training or by inference. All learning networks where the values of weights (and bias) are determined are called training. The training uses libraries of activation input data to compute weight values by generating output predictions, and it can also adjust the weight values to get better output predictions. Neural network inference is an approach to run a program using the determined weight values obtained during the training process to calculate the output of the network. The most important thing to remember is that all processors designed for training can also perform well in inference, but the opposite is not always true. Also, there are many ways to compute (train) the weights in the neural networks, but the most common approaches are shown in Figure 2.5 [41].

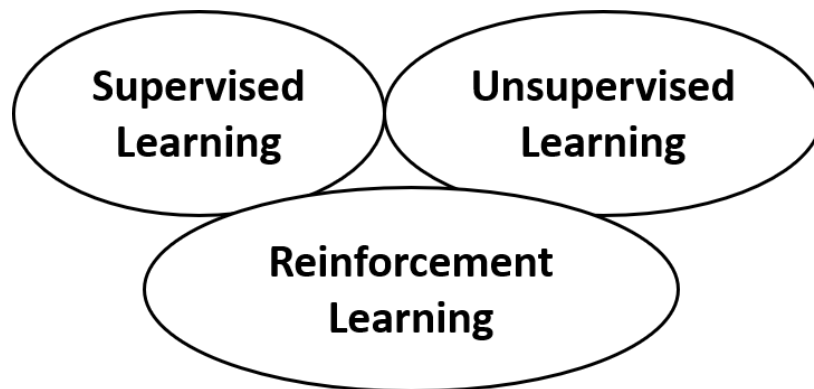


Figure 2. 5 Type of Learning in Artificial Neural Network

Supervised Learning is a learning algorithm in which all training samples are given labels to predict the outcome of the given tasks, and is also called classification of a data point.

Unsupervised Learning: Unlike supervised learning, labels do not need to be mapped in all training samples to find patterns, structures, or clusters in the data. Unsupervised learning algorithms are often used for analysis and visualization processes in the project, where the goal is to extract low-

dimensional data from high-dimensional data. Reinforcement learning is an algorithm technique used to train weights according to the state of the current environment by using an agent that learns the behaviour of the given environment and performs actions to predict the outcome. The agent receives a reward for the correct response, which increases the confidence in the processes for further actions. An example of reinforcement learning is Markov Decision Process (MDP).

There are many specific algorithms used in the learning process, but Table 2.3 summarizes all the common algorithms that are popular in machine learning domains due to their better performance.

Table 2. 3 Common algorithms in machine learning [45][46]

Category	Name of Algorithms
Traditional Machine Learning Algorithms	Decision trees
	Naive Bayes
	Random forest
	Support vector machine
	K-nearest neighbor
	K-means clustering
	Gaussian mixture model
	Hidden Markov model
Deep Neural Networking Learning Algorithms	Markov Decision Process (MDP)
	Data Dependency
	Support Vector Machine (SVM)
	Local Binary Patterns (LBP)
	K-Step Contrastive Divergence for RBMs
	Greedy layer-by-layer algorithm for DBN
	Autoencoder (AE)
	Recurrent Neural Network
	Convolutional Neural Network
	Long short-term memory
	Multilayer Perceptron (MLP)

2.1.3.1 Convolutional Neural Networks (CNN)

The first DNN model to become famous was LeNet [37], a Convolutional Neural Network (CNN) used to recognize handwritten digits. CNN is the most commonly used for object recognition which became popular in 2012 when AlexNet won the ILSVRC competition by outperforming the earlier methods [44][47]. Since that time, The DNN models have become increasingly complex and accurate, due to the increasing availability of computational hardware and memory resources. Table 2.1 summarizes the history of the CNN model.

Convolutional Neural Networks (CNN) Algorithm composed of the featuring learning and classification part of an input to predict an output. Featuring learning is the main part that is used to extract all the important features in the input by using kernel and pooling. These layers are repeated two times or more than two to form deep learning by having more than one hidden layer.

Figure 2.6 shows the structure of CNN algorithms.

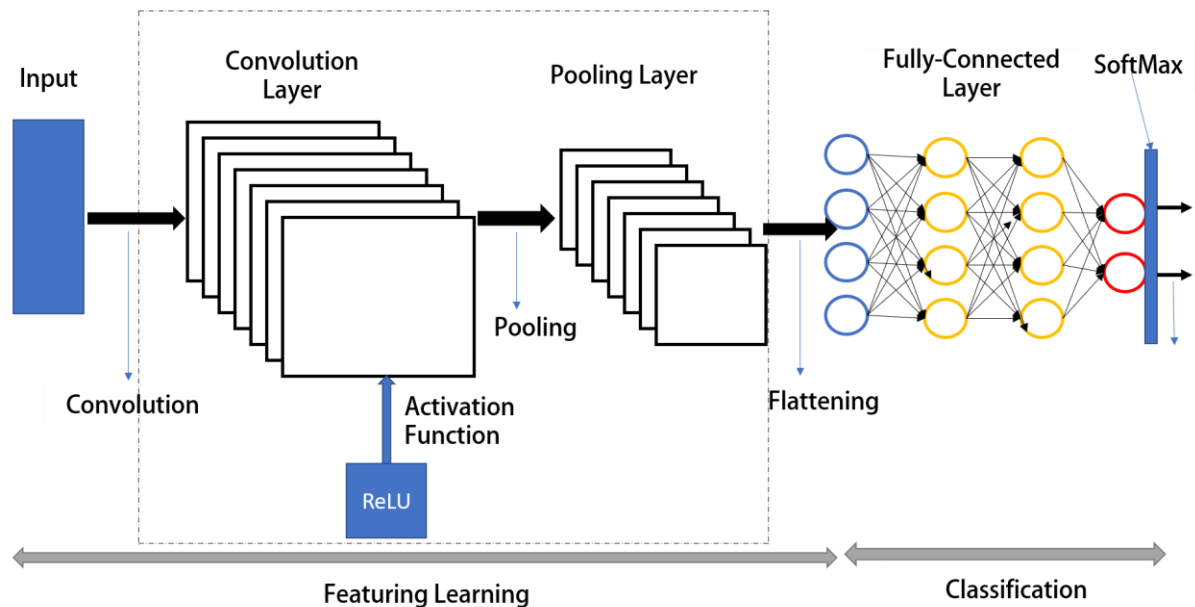


Figure 2. 6 Convolutional Neural Network (CNN) Algorithms

Table 2. 4 History of the CNN Models

Year: Name of Model	Description
1998: LeNet [37]	First CNN model for handwritten digits
2012: AlexNet [44]	The first model of CNN won ILSVRC
2014: VGG16 [48]	Contribute Smaller kernel Size (3x3) by increasing depth using an architecture.
2015: GoogLeNet [28]	Improved utilization of the computing resources inside the network (Inception block)
2015: Inception v3 [49]	Exploring ways to scale up networks for efficient computation (i.e., factorized convolutions)
2016: Inception v4 [50]	Improve inception blocks
2016: ResNet [51]	Optimize a residual network learning by skip connections
2017: Xception [52]	Pointwise and depth wise convolutions
2017: ResNetXt101_64x4d [53]	Investigate ResNeXt on an ImageNet-5K set (Grouped convolution)
2017: DenseNet161[54]	Data flows across layers
2018: SeNet154 [55]	Enhancing spatial encoding in the channels by exploiting dependencies between them
2018: NasNet-A [56]	Contribute search engine in Neural Architecture by using transfer learning
2019: BERT-L [57]	Transformer model for Natural Language Processing (NLP)
2019: Megatron [58]	Parallel transformer model for NLP

Featuring Learning in CNN is the extraction process of the important part in the input image into smaller reshapes to simplify the performance of learning. Classification is the ability to identify input by comparing it with the original label dataset. The process of reduction and learning in each layer (i.e., Convolutional, Pooling, Flattening layers) is managed in Equation 2.3.

$$OUTPUT\ RESHAPE = \frac{INPUT - KERNEL\ SIZE + (2 * PADDING)}{STRIDE} + 1 \quad (2.3)$$

Where: *INPUT*: Size of the input if 32 x 32 will be 32, *KERNEL SIZE*: filter size if 5x5 will be 5, *PADDING*: Adding matrix size, and *STRIDE*: How too far to move the filter/kernel.

The reshape size in Equation 2.3 helps to calculate the number of training parameters in the performance of the CNN algorithms. These parameters determine how better the algorithms perform learning by depends on the size of filters and input for each layer. Equation 2.4 expresses the relationship of the number of parameters as follows:

$$Number\ of\ Parameters = Nf(t) * [(Sf(t) * Sf(t - 1)) + 1] \quad (2.4)$$

Where: *Nf(t)*: Number of filters of the current layer, *Sf(t)*: Size of Current filter, and *Sf(t-1)*: Size of the previous filter.

The structure in Figure 2.6 involves seven processes which are input, convolution, pooling, flattening, fully-connected, SoftMax, and output. The following is the description of each operation within CNN algorithms.

2.1.3.1.1 Convolution Input

Convolution Input is any matrix or vector of data for either objects or text or speeches that require classification or prediction in a particular output in better performance and accuracy. The input

can be an image(s), sound(s), video(s), text(s), or handwritten digits that require unique interpretation during the training process. CNN accepts input that can be classified to obtain the output. The common CNN benchmark datasets are MNIST, CIFAR-10, and ImageNet. MNIST is a dataset of grayscale images of a handwritten single digit between 0,1,2... to 9 with a size of 28x28 pixels. Figure 2.7 shows an example of the digits in the MNIST dataset [59].



Figure 2. 7 MNIST Benchmark [59].

CIFAR-10 is the modification of MNIST to accept color photographs objects with the size of 32x32 pixels as shown in Figure 2.8. Both MNIST and CIFAR-10 contained the same number of samples about 60,000 [60].

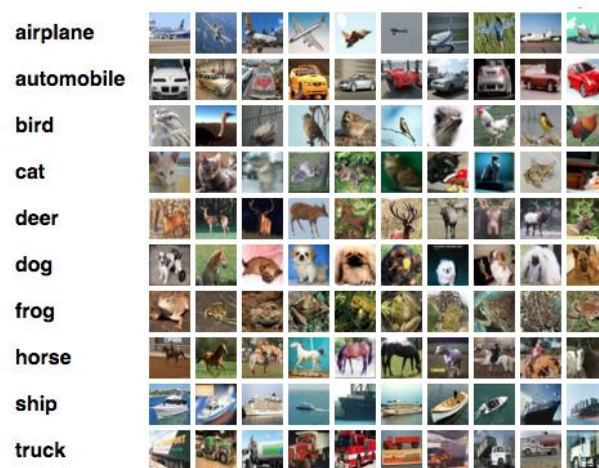


Figure 2. 8 CIFAR-10 Benchmark [60].

ImageNet is the latest dataset of CNN which contains more than 14 million images with the size of 224x224 pixels as shown in Figure 2.9. It consumes a large amount of computing power in terms of GPU, CPU, and RAM [61].



Figure 2. 9 ImageNet Dataset Benchmark [62]

2.1.3.1.2 Convolutional Layer

The convolutional layer is the dot product between the input image and the kernel used to divide the input into small sizes. The layer takes the given dimensional input layer to extract the features in the input using the filter(kernel). The output of the computation is called a feature map, which has a different size than the input, as explained in Equation 2.3. The output can also be controlled by implementing padding, stride, and depth. The kernel convolves with the input by multiplying its elements by a set of weights. Equation 2.5 expresses the convolution operation.

$$f^k = \sum_c^n \sum_{x,y}^n I_c(x,y) * e^k(u,v) \quad (2.5)$$

Where:

f^k : feature map value at k slice

$I_c(x, y)$: Slice of the input at a matrix of size x, y

$e^k(u, v)$: Kernel size at u, v

Figure 2.10 demonstrates equation 2.5 on how the convolution layer computed the feature map.

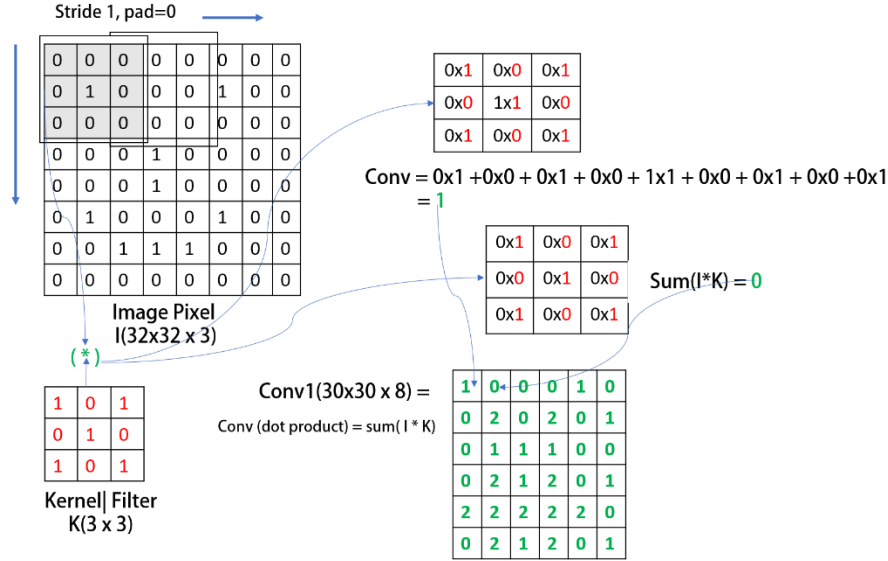


Figure 2. 10 Demonstration of Convolution Computation in CNN for One Channel

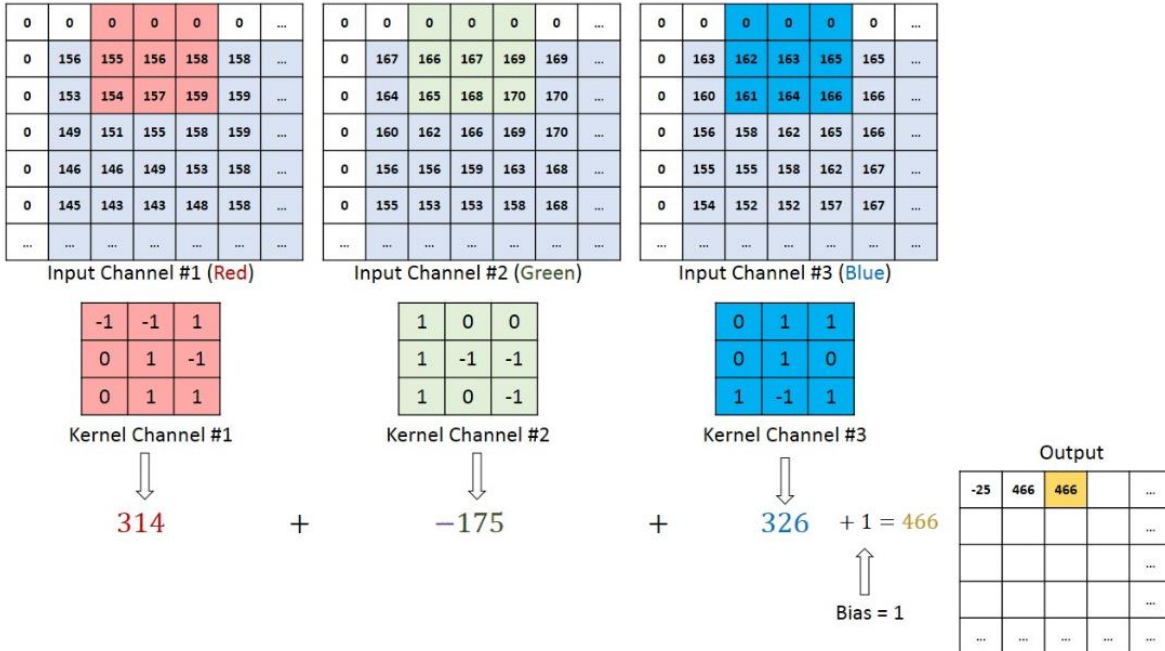


Figure 2. 11 Demonstration of Convolution Computation in CNN for Three Channels [63]

However, [64] explains the simplification way of the process of convolution in hardware just by matrix multiplication between input and kernel after converted into space matrix as shown in Figure 2.12.

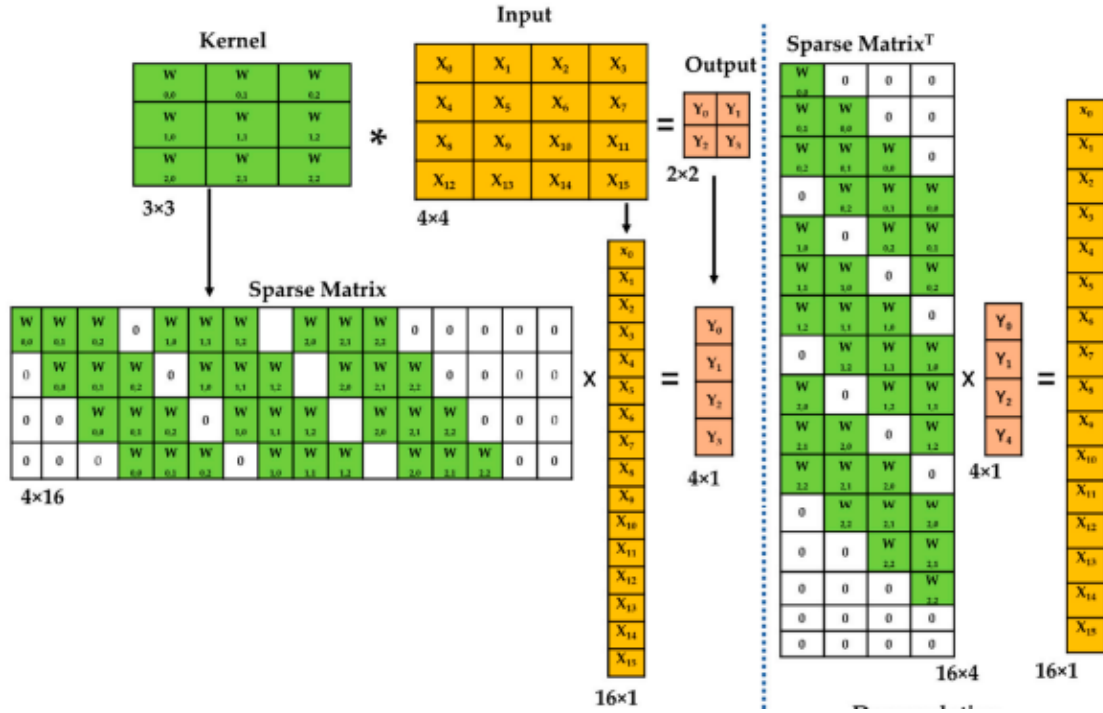


Figure 2. 12 The cracking of Convolution and Deconvolution Operation [64]

2.1.3.1.2 Pooling Layer

The pooling layer provides an approach to down sampling feature maps by pooling the presence of features into patches of the feature map. There are two common types of pooling, max-pooling, and average-pooling. In max-pooling, the maximum value in the input is calculated according to the pooling size, while in average-pooling, the average of the convolutional outputs is calculated according to the pooling size. Pooling is commonly used to reduce the dimension of the input by

grouping all the important features. Figure 2.13 demonstrates the calculation of pooling in the CNN.

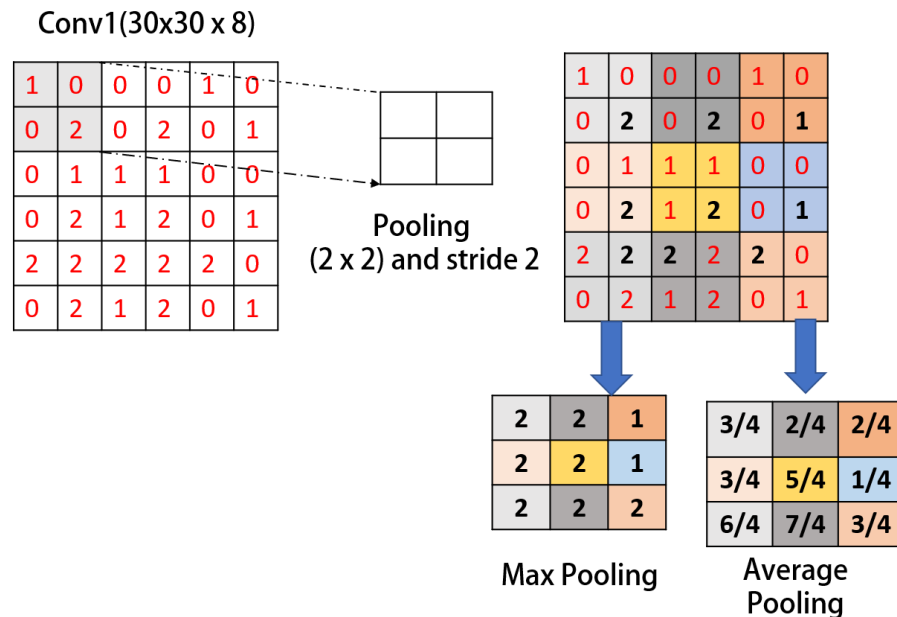


Figure 2. 13 Demonstration of Pooling Computation

2.1.3.1.3 Flattening

The flattening converts the pooling feature map output from 2D to 1D by changing all rows to a single column. The result of flattening is the input for the fully connected layer to start the learning process of the artificial neural network by adjusting the weights and learning rate. Figure 2.14 illustrates the flattening process in the CNN.

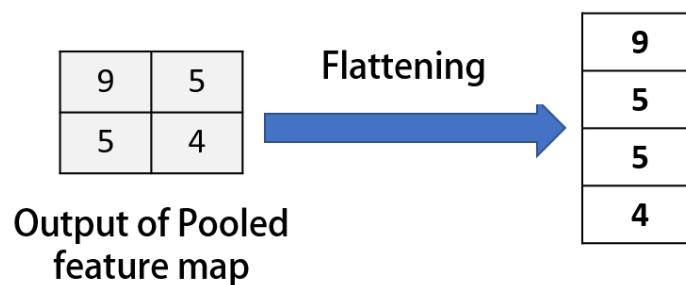


Figure 2. 14 Flattening Operation

2.1.3.1.4 Fully-Connected Layer

This is the important layer to train the CNN algorithms by computing the desired output from the flattening input and the hidden layers shown in Figure 2.15. The fully-connected computation includes forwarding propagation and backpropagation, where forwarding propagation is the dot product of the input with its weight in the forward direction as shown in Equation 2.6:

$$y_i = f\left(\sum_{i=0}^n (x_i w_i) + b\right) \quad (2.6)$$

Where: y_i : Output of the layer, f : Activation function, x_i : Input value at I , W_i : Weight of the Signal at I , and b : Bias of the layer

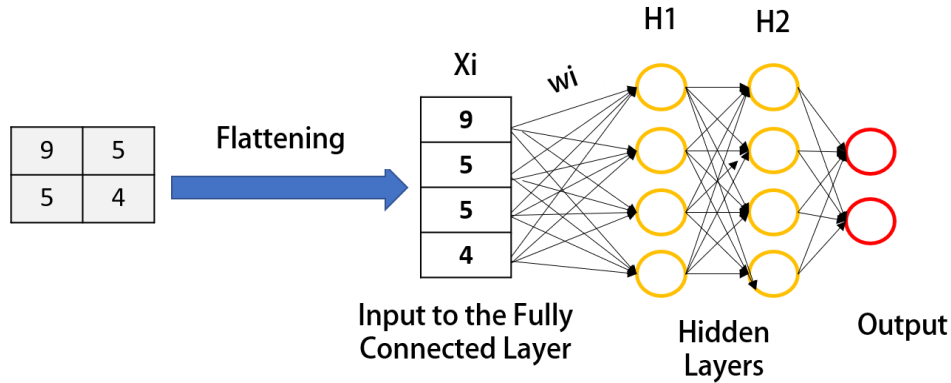


Figure 2. 15 Sample of Fully-Connected Layer

Backpropagation in the fully connected is used to update the weight and bias in the connection to improve the accuracy of artificial learning. The stochastic gradient descent algorithm is used during training by updating the weight and backpropagation to approximate the errors of the training samples. The following are the equations for stochastic gradient descent [65]:

$$W_{new}^i = W_{old}^i - \alpha \frac{\partial l}{\partial W^i} \quad (2.7)$$

Where: α is the learning rate, W_{new} : weight co-efficient values, W_{old} : weight old co-efficient value, $\frac{\partial l}{\partial w}$ is the rate of error for weight i .

$$b_{new}^i = b_{old}^i - \alpha \frac{\partial l}{\partial b^i} \quad (2.8)$$

Where: α is the learning rate, b_{new} : bias co-efficient values, b_{old} : bias old co-efficient value, $\frac{\partial l}{\partial b}$ is the rate of loss under bias.

The backpropagation equations from 2.9 to 2.12 have derived from partial derivative [65][66]:

$$\frac{\partial y_i}{\partial z_L} = \{ y(1 - Y) \text{ if } i = L \mid -yY \text{ if } i = L \} \quad (2.9)$$

$$\frac{\partial L}{\partial z_L} = (y - Y) \quad (2.10)$$

$$\frac{\partial L}{\partial w_i} = -(1 - y_i)(x_i) \quad (2.11)$$

$$\text{Cross Entropy Loss} = - \sum_{i=1}^c Y_i \log(y_i) \quad (2.12)$$

Where: Y : Actual Output (Label value in the dataset), y : Desire Output from Computation, x : Input Value, and L : Loss.

2.1.3.1.5 SoftMax

The SoftMax layer is the layer in CNN that works by converting the fully-connected output value into probability so that it is easy to predict the desired output or compare it to the actual value. The

process is performed by the SoftMax operation as part of the sigmoid activation function in artificial machine learning. Equation 2.13 represents the SoftMax operation as follows:

$$P(z) = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}} \quad (2.13)$$

Where: $P(x)$: Probability of the output z
 Z_i : Output at i
 $\sum_{j=1}^k e^{z_j}$: Total exponential value of outputs

The probability value of the desired output (y) helps the machine learning process to verify the accuracy of the output by determining the cross-entropy loss, as shown in Equation (2.12). If the obtained cross-entropy loss is high, it means that the learning should go backpropagation to adjust the weighting until an acceptable accuracy of the error and at the same time, the number of training parameters calculated.

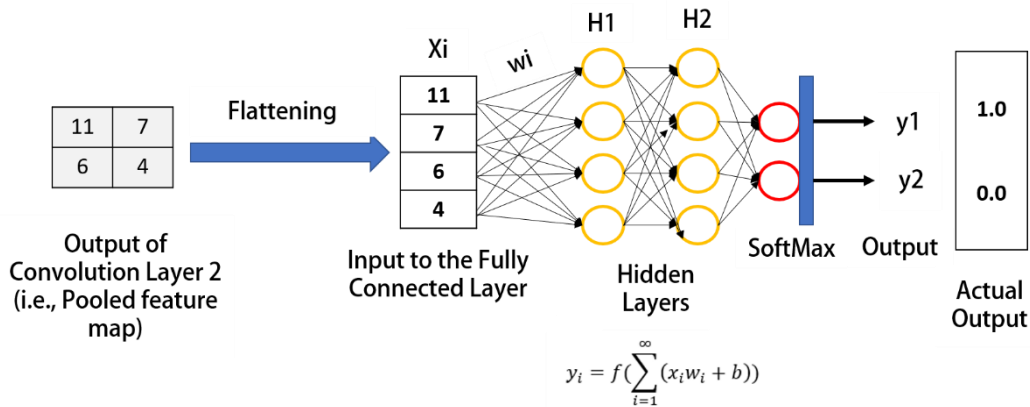


Figure 2. 16 Fully Connected Layer and SoftMax

2.1.3.2 Recurrent Neural Network (RNN)

The RNN model was briefly introduced by Hopfield Newark in 1982, after having been developed in 1980. The idea of the model is to implement a sequence of input vector and output vector over time, such as understanding text or speech, or videos [67]. A recurrent neural network (RNN) is

another deep neural network algorithm that can share parameters across layers by inputting the previous output to the next stage. Figure 2.17 shows the structure of the RNN model.

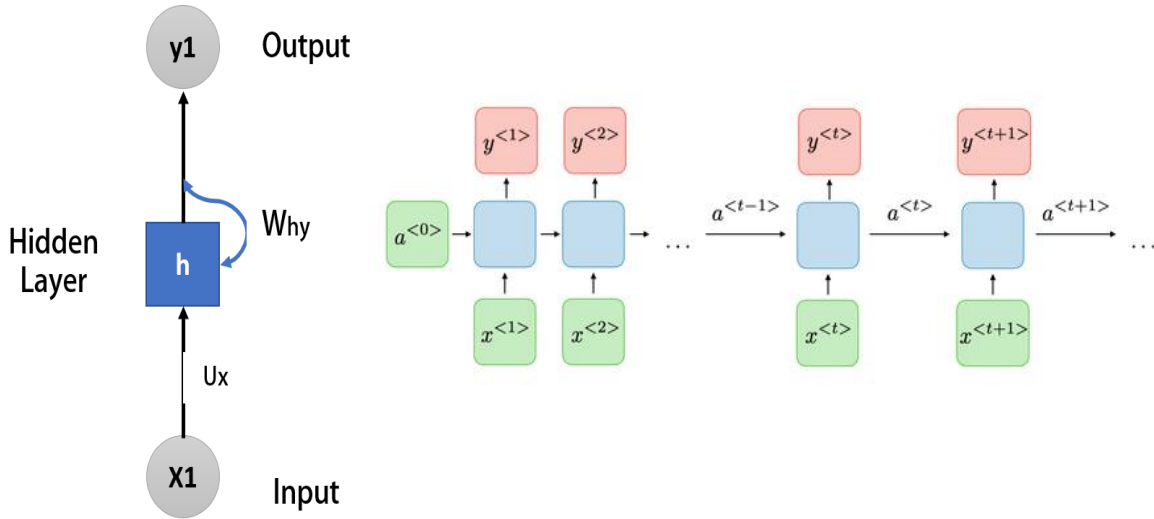


Figure 2. 17 Fold and Unfold Structure of the RNN Model [68]

The training parameters depend on the time step t , so they all expressed as a function of t as shown in Figure 2.17 as input x^t and output y^t . The relationship between inputs and outputs is connected according to Equations 2.8 and 2.8.

$$a^{(t)} = g1(ba + W_{aa}a^{(t-1)} + W_{ax}x^{(t)}) \quad (2.14)$$

$$y^{(t)} = g2(by + W_{ya}a^{(t)}) \quad (2.15)$$

Where: g_1 and g_2 are activation functions, W_{aa} and W_{ay} are weight matrix, and by and ba are biases values.

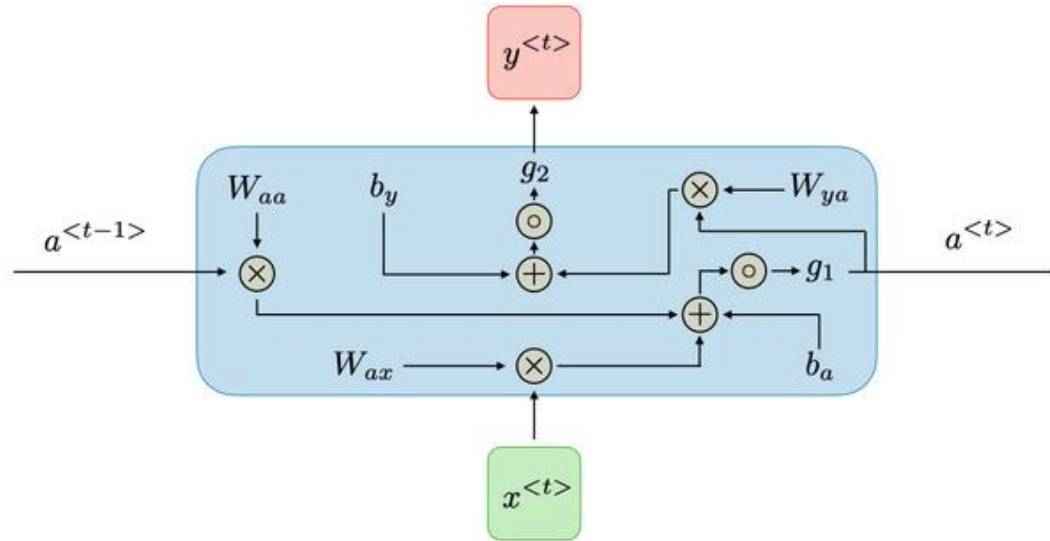
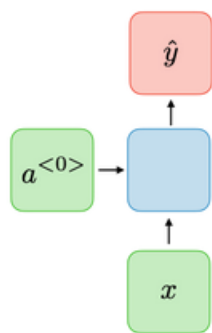
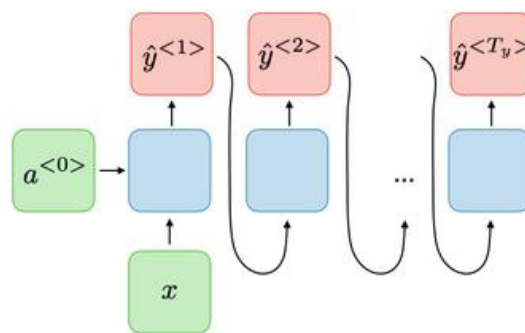


Figure 2. 18 RNN operations and Parameters [68]

Sigmoid, Tanh, and ReLU are common activation functions implemented in the RNN model, which is the same as other deep learning algorithms. The RNN model is popular for many applications nowadays, but [68][69] researchers criticize that it is slow in computation and cannot access historical information from a long time. RNN are divided into four categories known as one-to-one, one-to-many, many-to-one, and many-to-many, and each depending on the type of application in natural language processes. The one-to-one model is a commonly used traditional neural network, the one-to-many model is used for music generation, the many-to-one model is used for sentiment classification, and the many-to-many model is used for machine translation and name and entity recognition.



a.) One-to-One



b.) One-to-Many

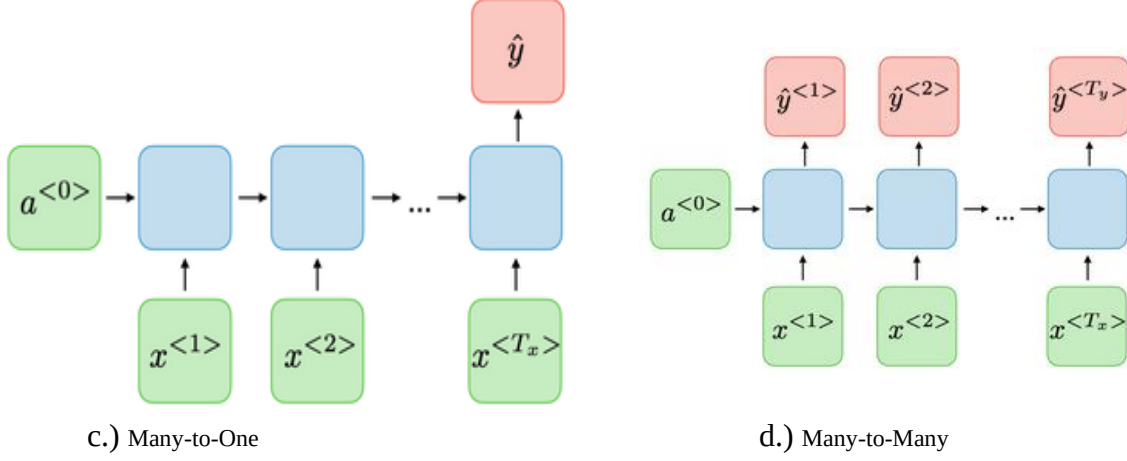


Figure 2. 19 Categories of RNN Model [68]

Training the RNN is performed by updating the weight parameters using a backpropagation algorithm that depends on the output and its loss function, which expresses how far with an accuracy of the desired output. The loss function (ℓ) determines the deviation between the desired output and the actual output as shown in Equation 2.16. Equation 2.17 describes a change in the loss function, $(\partial L / \partial w)$ with respect to the weight (w) parameter [68].

$$\ell(y, Y) = \sum_{t=1}^T \ell(y^t, Y^t) \quad (2.16)$$

$$\frac{\partial L}{\partial w} = \sum_{t=1}^{T_y} \frac{\partial L}{\partial w} \bigg| (t) \quad (2.17)$$

2.1.3.2.1 Type of Gates in RNN

Traditional RNN is facing problem to handle long time dependencies especial the changing of weight which either it too small which shrink activation to zero along with the layers (vanishing gradient) or it too large random weights multiply along with layers (exploding gradient). With that situation, [70] proposed Long Short Time Memory (LSTM) RNN model to figure out the

challenges. LSTM is made up of a cell, and three gates (input, output, and forget) whereby a cell memorize values over inconsistent intervals of time while gates manage information passing into and out the cell. LSTM is performing a revolution in speech recognition applications since 2007 [71] which has able to handle large-vocabulary speeches and text to speech which achieved be implemented in many applications such as Google's Speech recognition [30][72], and Machine translation [73].

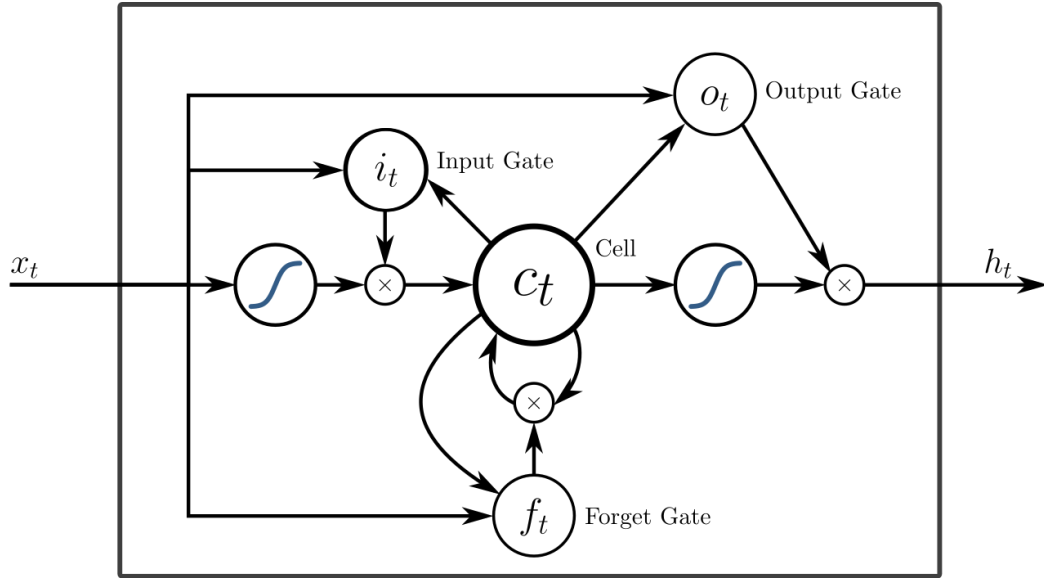


Figure 2.20 Long Short Time Memory Dependencies Block Diagram [74]

Where: The forget gate manages to erase the cell, and the output gate manages the amount of reveal in the cell. The following are equations represent the implementation of the dependencies in LSTM [70][75]:

$$f^{(t)} = g(bf + W_f x^{(t)} + U_f h^{(t-1)}) \quad (2.18)$$

$$i^{(t)} = g(bi + W_i x^{(t)} + U_i h^{(t-1)}) \quad (2.19)$$

$$o^{(t)} = g(bo + W_o x^{(t)} + U_o h^{(t-1)}) \quad (2.20)$$

$$c'^{(t)} = \tanh(bc + W_c x^{(t-1)} + U_c h^{(t-1)}) \quad (2.21)$$

$$c^{(t)} = (f^t \odot c^{(t-1)} + i^t \odot c'^{(t)}) \quad (2.22)$$

$$h^{(t)} = o^t \odot g(c^{(t)}) \quad (2.23)$$

Where: f^t is the forget gate, i^t is the input gate vector, o^t is the output gate vector, h^t is the hidden layer vector, g is the sigmoid activation function, c^t is the cell vector, and $\odot$ is the element-wise product notation known as Hadamard Product, $h_0 = 0$ and $c_0 = 0$.

Gated Recurrent Unit (GRU) is also another solution of vanishing gradient and exploiting gradient in the traditional RNN was introduced in 2014 [76] for applications such as natural language processing, music models, and speech signal modeling [77][78]. GRU reduces the number of parameters in LSTM for better performance on certain less frequency and small datasets [79].

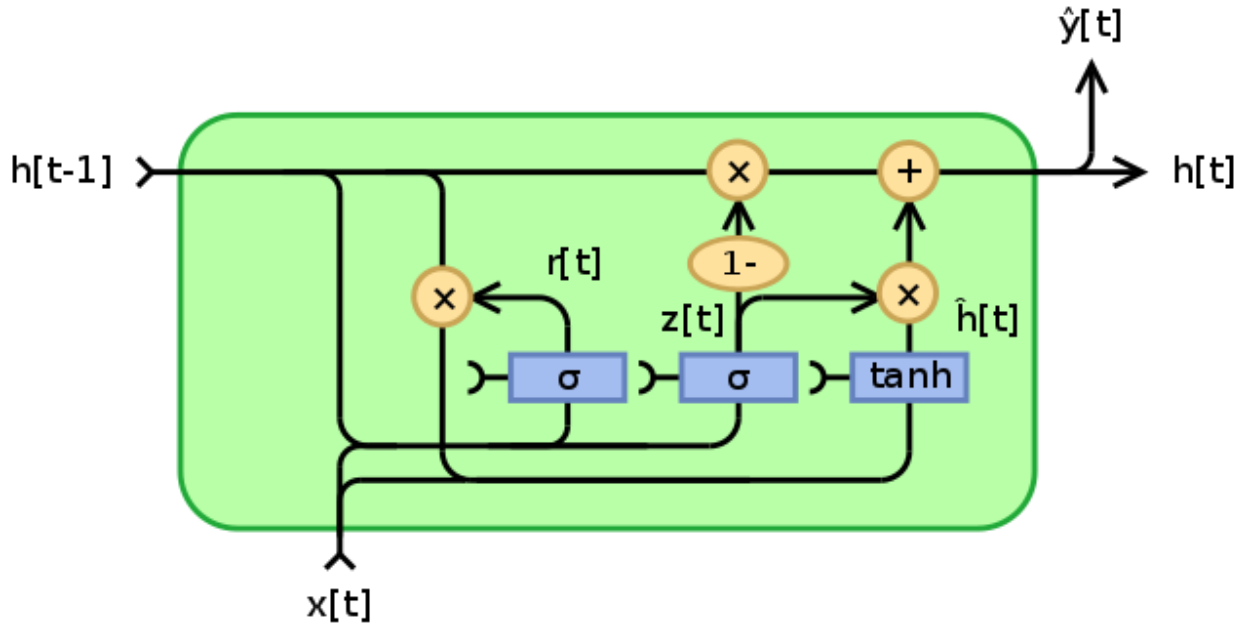


Figure 2. 21 Gated Recurrent Unit (GRU) [79]

Where:

$$z^{(t)} = g(bz + W_z x^{(t)} + U_z h^{(t-1)}) \quad (2.24)$$

$$r^{(t)} = g(br + W_r x^{(t)} + U_r h^{(t-1)}) \quad (2.25)$$

$$h'^{(t)} = \tanh(bh + W_h x^{(t)} + U_h (r^{(t)} \odot h^{(t-1)})) \quad (2.26)$$

$$h^{(t)} = (1 - z^{(t)}) \odot h^{(t-1)} + z^{(t)} \odot h'^{(t)} \quad (2.27)$$

Where: $x^{(t)}$ is the input vector, $h^{(t)}$ is the output vector, $h'^{(t)}$ is the candidate activation vector, $z^{(t)}$ is the update gate vector, $r^{(t)}$ is the reset gate vector, W is the input weight vector, U is the hidden weight vector, and b is the biases.

2.1.3.2.2 Word Representation Techniques in RNN

Most machine algorithms require all input and output variables to be in numeric form because they cannot work directly with label data or categorical data. Categorical data is the values of the label variables instead of numeric values, such as: "colour" variable with values of {"red", "blue", "black" ...}. RNN belongs to the algorithms that require the numerical representation of variables, which can be represented in two ways: 1-Hot Representation and Word Embedding [68].

2.1.3.2.2.1 1-Hot Representation

1- Hot Representation is the machine learning technique to encode the input variable into a binary vector. It assigns binary value 1 to the correct position of the word in the set and sets the rest bits to zeros [80][81]. For example, there are two sentences, namely "Have a great day" and "Have a good day", which need to be checked for their similarity in meaning. Then each word is varied in a label input matrix $E = \{\text{have, a, good, great, day}\}$, which can be represented in 1-hot coding as shown in Table 2.5. It is easy to implement in machine learning, which has the advantage of being much easier to implement in FPGA [82][83].

Table 2. 5 Sample of 1-Hot Representation

Word in Matrix, E {Have, a, good, great, day}	1-Hot Vector Representation
Have	[1, 0, 0, 0, 0]
a	[0, 1, 0, 0, 0]
good	[0, 0, 1, 0, 0]
great	[0, 0, 0, 1, 0]
day	[0, 0, 0, 0, 1]

2.1.3.2.2.2 Word Embedding

Word Embedding is the representation of words into a real value vector that implies the meaning of the words or the similarity in the meaning of the word. This is a popular technique that has been implemented in many types of research for natural language processing applications [84]. The main goal of word embedding is to identify and classify the semantic similarities based on their properties in the linguistic elements from a sample of the large dataset [85].

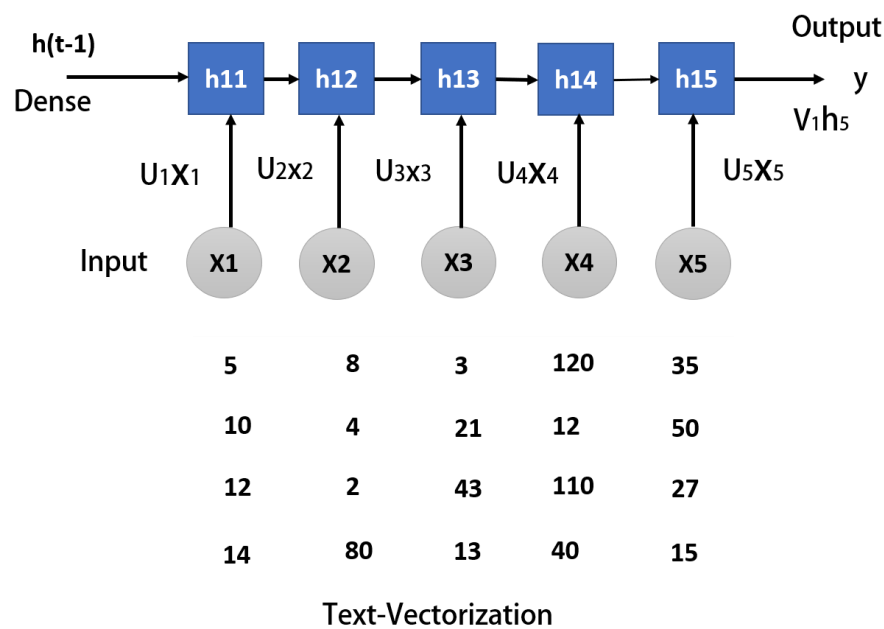


Figure 2. 22 Sample of Word Embedding

Word Embedding has been extended with subcategories such as skip-gram, Word2vec, and multi-sense skip-gram (MSSG) to achieve better performance on multi-sense embeddings by representing a word according to knowledge-based and unsupervised learning [86][87]. Word2vec is a learning framework for embedding target words by estimating the nearest (i.e., probability) with surrounded other words. The skip-gram word2vec performs supervised word embedding by accessing the probability between the given target word and the context word [68]. Negative Sampling is the more efficient word embedding by using binary classifiers to evaluate the

probability of how far between the given target word and the context words are likely to appear simultaneously [88]. The comparison between words is calculated using the method Cosine Similarities as shown in Equation 2.27 and Figure 2.23.

$$Similarity = \frac{w1 \cdot w2}{|w1||w2|} = \cos(\theta) \quad (2.27)$$

Where: $w1$ is the first word, $w2$ is the second word, and θ is the angle between $w1$ and $w2$

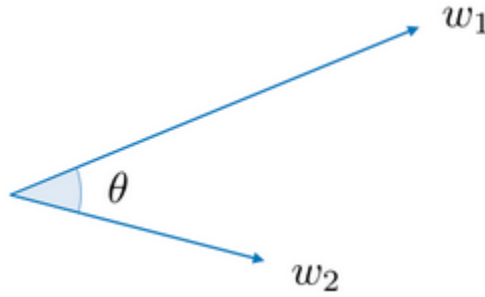


Figure 2. 23 Angle of Similarity between the words [66]

In summary, the chapter presented the background information about DNNs on how it works and its position in artificial intelligence. The chapter also mentioned the different algorithms such as Multilayer Perceptron (MLP), Convolution Neural Network (CNN), Recurrent Neural Network (RNN), etc., that used nowadays according to various applications. With that algorithms, the CNN and RNN algorithms are selected to be implemented in this work. Then CNN and RNN algorithms described in more detail included their structures, tools, and datasets.

CHAPTER 3

LITERATURE SURVEY

This chapter describes a literature review of various architectures that have been proposed for the execution of deep learning (DL) and neural network (NN) algorithms. The chapter briefly discusses each of the recent architectures proposed in the literature, highlights the strengths and limitations of each approach, and attempts to identify research gaps that can be addressed in this study. An overview of machine learning architectures is provided in Section 3.1. The common CNN architectures discussed in Section 3.2 and Section 3.3 demonstrate some of the proposed machine learning instructions set.

3.1 Overview of Machine Learning Architecture

Traditional architectures such as CPUs and GPUs, described as general-purpose architectures, have been used to accelerate the execution of deep learning and neural network algorithms [16][88][89]. However, these architectures are not performed well due to their need to use a large number of hardware resources. To address these issues, hardware accelerators such as Field Programmable Gate Array (FPGA), Application Specific Integrated (ASIC), and special domain architecture have been proposed to achieve the efficiency that was missing in general-purpose architectures [90].

In [20] the extension of general-purpose architecture by using hardware accelerators is explained as they are most efficient in terms of energy and performance when optimized for a specific application. The authors of the paper contribute a new architecture by combining a host processor and a flexible design paradigm for accelerators that can easily adapt neural network structures and

their algorithms. The architecture consists of a host processor, two-level caches (L1 and L2), the main memory, and an integrated accelerator that contains two memories (i.e., a private memory and a shared memory) and a DMA operation to the main memory. The shared memory of the accelerator is called scratchpad memory which is used to exchange data between the accelerator and the host CPU through two special instruction sets known as "load data" (sbLD) and "store data" (sbST), which help to isolate the sidebar from the main memory area while avoiding coherence problems. Finally, the architecture shows almost the same performance and energy compared to a fixed accelerator, despite its flexibility to handle many similar applications without developing a new accelerator, while a fixed accelerator required a new hardware accelerator for each application change. For all this, the architecture does not take into account the impact of DMA latencies at initial and final when streaming data to or from the accelerator, and it does not implement data allocation in the sidebar. It can also use scratchpad memory instead of L1 caches and high-levels to avoid miss-hits, which can help improve the overall performance of the architecture.

An example from previous research that has been implemented considers a hierarchical on-chip scratchpad memory SPM, called multi- TOPS AI [89]. The authors proposed a programmable architecture to accelerate deep learning training to achieve high performance, energy reduction, and area efficiency. The architecture consists of massive processing units structured as Two-Dimensional arrays with local register files, load and store units, and multi-level SPM. These units communicate with memory through the memory controller. The authors have introduced a custom instruction set to support their architecture.

Since many general-purpose architectures face the challenge of managing improper memory accesses from untrusted accelerators, [91] proposed a novel way to solve this problem by adding

a layer called Border Control between main memory and the accelerators' cache. Each memory access from the accelerators should pass through the boundary control layer to test and verify that it has permission to access memory. The boundary control layer consists of a protection table that contains all permission information and a small cache that stores the most recently used permission information. Thus, the memory access is protected from any kind of unauthorized access.

The simple and flexible user defined instruction (ISA) set called Cambricon for neural networks (NN) was presented by [88]. The proposed instructions consist of four types of instructions, namely data transfer, logic, computation, and control instructions. Some of these instructions have data-level parallelisms, such as vector and matrix instructions, where a single instruction can perform multiple operations (SIMD), which makes the code more compact compared to the traditional scalar and vector instructions. Another advantage of this architecture is the use of on-chip scratchpad memory as a replacement for vector registers so that a large amount of data can be loaded into scratchpad memory, which reduces memory access, which in turn contributes to significant power reduction and latency minimization. Moreover, the proposed architecture for this Cambricon instruction implementation has seven pipeline stages.

While many types of research on Deep Learning (DL) and Neural Network (NN) are applied to different domains and architectures, V. Vanhoucke, et al., [4] focused on modern conventional x86 CPUs by applying some approaches that can significantly improve the performance by a factor of 3 compared to floating-point baseline. The authors used SSE, which is an extension of Single Instruction Multiple Data (SIMD) by allowing instructions to process multiple data. They evaluate their approach using speech recognition as a benchmark.

The system of high-performance computing that can process and execute a very large NN of one billion with 3 machines within a few days is presented by [24]. The technology is called commodity off the shelf HPC and consists of many clusters of GPU servers and MPI is used to interconnect these GPU servers. Also, the system is scalable so that as the number of parameters increases, the number of machines also increases slightly. The disadvantage of this system is that it is not energy efficient as it requires a large number of hardware resources.

H. Esmailzadeh, et al, [92] proposed a programmable code level processor for neural networks in embedded systems called Neural network Stream processing core (NnSP) which can process a data stream by ordering the computation of a task into data streams that flow directly into massive processing units where computations are performed on these stream packets, processing units communicate over a bus where transactions are performed. The NnSP software supports the architecture in two ways: a stream compiler that receives the design requirements of the intended NnSP core on the one hand, and the topology with which many NNs are to be implemented with NnSP on the other hand, then the synaptic parallelization of data streams operating on the intended NnSP core is generated by the compiler output. The advantage of this stream compiler is that it allows the reuse of the NnSP core as well as the associated computational power. The second is an NnSP core builder that obtains the design requirements of the target processor, such as on-chip cache memory, buffer size, functional units, etc. In addition, simulation-generated results are passed to the core builder.

3.2 Neural Networks Architectures

The section describes the existed applied architectures that have been proposed to implement convolutional neural networks (CNN). The section briefly explains the details for each architecture including its applications and structure.

3.2.1 LeNet

LeNet is the first popular CNN architecture and was proposed by LeCuN in 1998 to be used for hand digit recognition [37]. This architecture then soon served as a kind of template for most modern CNN architectures nowadays [43]. The LeNet architecture essentially consists of several convolutional layers and average-pooling layers. It takes an input consisting of grayscale input images of size 28x28 for LeNet-1, and 32 x 32 pixels for LeNet-4, followed by a 5x5 convolution, then a pooling layer of 2x2, the last two steps are repeated, resulting in some fully connected layers. Finally, the output is a classification to classify the digits from 0 to 9 as shown in Figure 3.1. Due to lack of computational resources, this network was not able to process higher resolution images requiring more convolutional layers at this time.

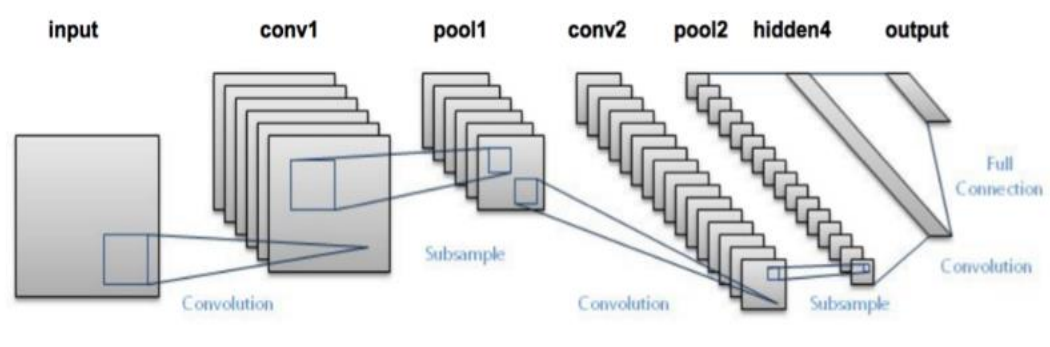


Figure 3. 1 LeNet Architecture [37]

3.2.2 AlexNet

This architecture is based on LeNet concepts. The shortcoming of LeNet is that it is not scalable for all image classes except handwritten digit recognition. Alexie was proposed by Krizhevsky [44] in 2012 and won by more than 10 percentage points over all competitors in ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), an annual computer vision competition. The architecture of the network is very similar to LeNet in terms of convolution and max-pooling operations, but then had some other innovations. AlexNet consists of convolution, max-pooling, dropout, data expansion, ReLU activations and has 60 million parameters. The network is feed forwarding with 3 layers more than LeNet. Hardware resource constraints at the time Alex was proposed led to the network being split and trained on 2 GPUs. AlexNet is deeper and wider than LeNet.

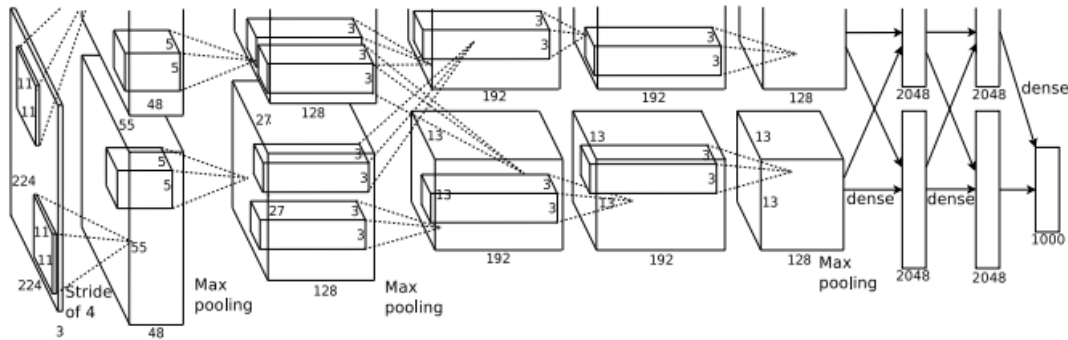


Figure 3. 2 AlexNet Architecture [44]

3.2.3 ZfNet

ZfNet is an improved version of AlexNet proposed by Zeiler and Fergus, hence the architecture was named after taken the first letter in their name (ZFNet) [44] [92]. It outperformed all previous Deep CNNs such as LeNet and AlexNet in terms of accuracy. ZfNet was the winner of ILSVRC in 2013, and the network uses a small filter size (7x7) compared to AlexNet (11x11). By using

such a smaller filter size, a lot of information and features are preserved, resulting in better accuracy.

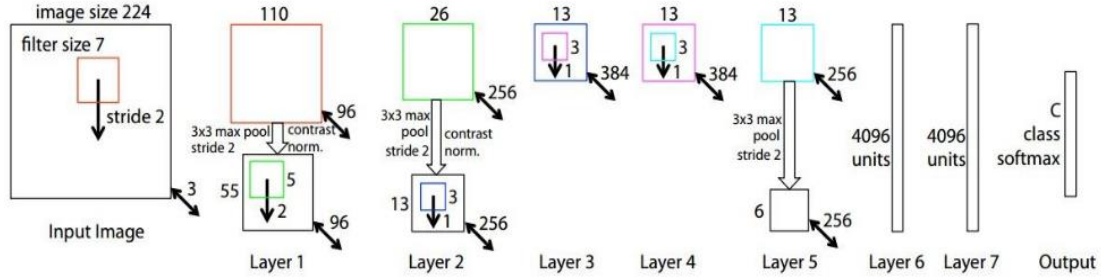


Figure 3. 3 ZfNet Architecture [43]

3.2.4 VGGNet

VGG was introduced by Karen Simonyan and Andrew Zisserman in 2013 [48] and was the runner-up in the 2014 ILSVRC competition. The network is characterized by having more depth compared to previous architectures, resulting in better performance, and being a unified architecture. VGG became the basis for many architectures developed later. It consists of 16 convolutional layers with 3 x 3 kernels, 11x11 filters in AlexNet, and 7x7 in ZFNet [2]. However, VGGNet filters can induce features as large-scale filters. This network has 138 million parameters which are challenging to handle.

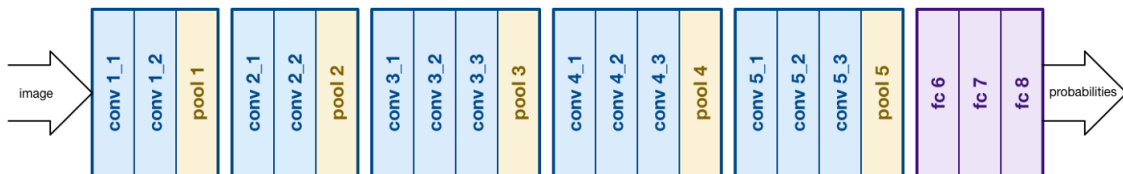


Figure 3. 4 VGGNet Architecture [43]

3.2.5 GoogLeNet

This network was developed by Google. GoogLeNet beat all other networks in the 2014 ILSVRC competition, including VGGNet, and became the winner [43][28]. It came with a new concept called Inception, which is the basic building block for the network, unlike previous architectures that used convolutional filters as the basic building block of the CNN. However, Inception includes a block of convolutional filters with different filter sizes. By using convolutional filters of different sizes, it can better capture objects at different scales. GoogLeNet uses small convolutional filters of size 1×1 , resulting in a significant reduction of parameters from 68 million to only 4 million parameters. The inception module is shown in Figure 3.5.

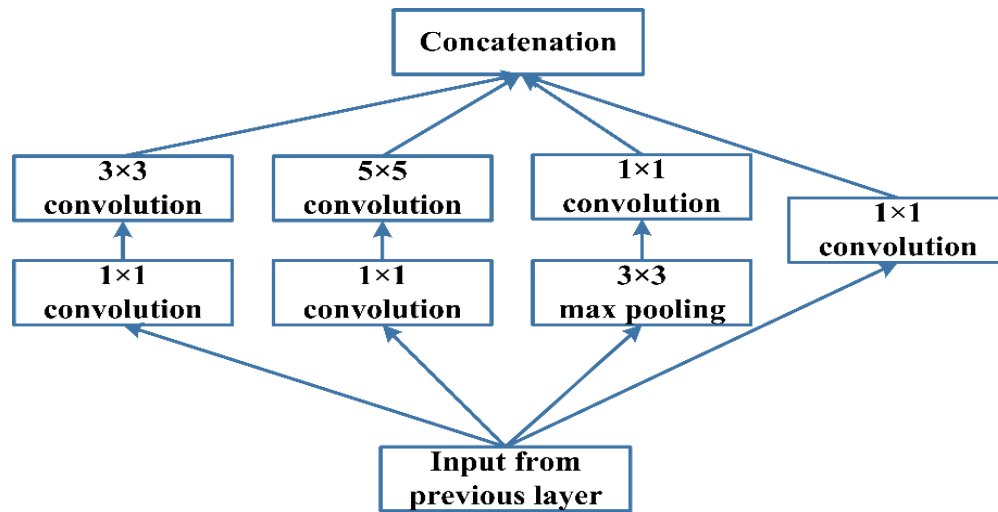


Figure 3. 5 GoogLeNet Architecture [28]

3.3 Neural Networks Instruction Set

CPU and GPU are among the common general-purpose processors typically used to run machine learning techniques, although they are not energy efficient due to the nature of the workload, which requires more flexible hardware resources. There is research that proposes a domain-specific

instruction set architecture for machine learning. In this section, we survey some of the existing Instructions Set for machine learning.

In the work [93], a novel domain-specific instruction set architecture (ISA) is proposed for Cambricon Architecture as Neural Network accelerator, which is also known as load-store architecture. The architecture consists of matrix, vector, scalar, logic, control, and data transfer instructions for NN with 64 bits and 64-32 bits for general purpose registers as shown in Table 3.1. In research, the instructions were extended to target large-scale machine learning, which showed better performance than general-purpose ISAs such as GPGPU, x86, and MIPS.

Table 3.1 An Overview to Cambricon Instructions [93]

Instruction Type		Examples	Operands
Control		jump, conditional branch	register (scalar value), immediate
Data Transfer	Matrix	matrix load/store/move	register (matrix address/size, scalar value), immediate
	Vector	vector load/store/move	register (vector address/size, scalar value), immediate
	Scalar	scalar load/store/move	register (scalar value), immediate
Computational	Matrix	matrix multiply vector, vector multiply matrix, matrix multiply scalar, outer product, matrix add matrix, matrix subtract matrix	register (matrix/vector address/size, scalar value)
	Vector	vector elementary arithmetics (add, subtract, multiply, divide), vector transcendental functions (exponential, logarithmic), dot product, random vector generator, maximum/minimum of a vector	register (vector address/size, scalar value)
	Scalar	scalar elementary arithmetics, scalar transcendental functions	register (scalar value), immediate
Logical	Vector	vector compare (greater than, equal), vector logical operations (and, or, inverter), vector greater than merge	register (vector address/size, scalar)
	Scalar	scalar compare, scalar logical operations	register (scalar), immediate

The extension of the instructions in the article [93] is Matrix-Multi-Vector (MMV) instruction for the feedforward computation, Vector-Multi-Matrix (VMM) for the backpropagation training process, Matrix-Multi-Scalar (MMS) instruction, a Matrix-Add-Matrix (MAM) instruction to perform the weight update. Pooling processes were also proposed for executing the Vector Greater

Than Merge (VGTM) instruction. Figures 3.6 and 3.7 show the format of the proposed instruction set for the Cambricon architecture.

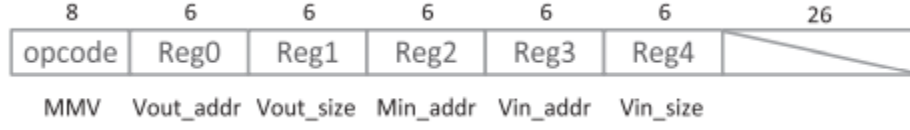


Figure 3. 6 Matrix Multi-Vector (MMV) Instruction [93]

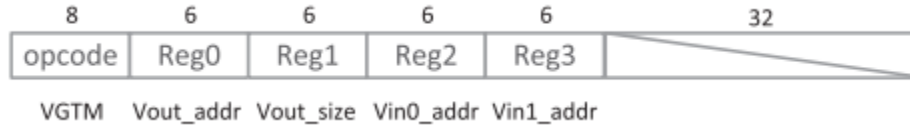


Figure 3. 7 Vector Greater Than Merge (VGTM) instruction [93]

MLP code: <pre>// \$0: input size, \$1: output size, \$2: matrix size // \$3: input address, \$4: weight address // \$5: bias address, \$6: output address // \$7-\$10: temp variable address VLOAD \$3, \$0, #100 // load input vector from address (100) MLOAD \$4, \$2, #300 // load weight matrix from address (300) MMV \$7, \$1, \$4, \$3, \$0 // Wx VAV \$8, \$1, \$7, \$5 // tmp=Wx+b VEXP \$9, \$1, \$8 // exp(tmp) VAS \$10, \$1, \$9, #1 // 1+exp(tmp) VDV \$6, \$1, \$9, \$10 // y=exp(tmp)/(1+exp(tmp)) VSTORE \$6, \$1, #200 // store output vector to address (200)</pre>	Pooling code: <pre>// \$0: feature map size, \$1: input data size, // \$2: output data size, \$3: pooling window size - 1 // \$4: x-axis loop num, \$5: y-axis loop num // \$6: input addr, \$7: output addr // \$8: y-axis stride of input VLOAD \$6, \$1, #100 // load input neurons from address (100) SMOVE \$5, \$3 // init y L0: SMOVE \$4, \$3 // init x L1: VGTM \$7, \$0, \$6, \$7 // V feature map m, output[m]=(input[x][y][m]>output[m])? // input[x][y][m]:output[m] SADD \$6, \$6, \$0 // update input address SADD \$4, \$4, #1 // x++ CB #L1, \$4 // if(x>0) goto L1 SADD \$6, \$6, \$8 // update input address SADD \$5, \$5, #1 // y++ CB #L0, \$5 // if(y>0) goto L0 VSTORE \$7, \$2, #200 // store output neurons to address (200)</pre>	BM code: <pre>// \$0: visible vector size, \$1: hidden vector size, \$2: v-h matrix (W) size // \$3: h-h matrix (L) size, \$4: visible vector address, \$5: W address // \$6: L address, \$7: bias address, \$8: hidden vector address // \$9-\$17: temp variable address VLOAD \$4, \$0, #100 // load visible vector from address (100) VLOAD \$9, \$1, #200 // load hidden vector from address (200) MLOAD \$5, \$2, #300 // load W matrix from address (300) MLOAD \$6, \$3, #400 // load L matrix from address (400) MMV \$10, \$1, \$5, \$4, \$0 // Wv MMV \$11, \$1, \$6, \$9, \$1 // Lh VAV \$12, \$1, \$10, \$11 // Wv+Lh VAV \$13, \$1, \$12, \$7 // tmp=Wv+Lh+b VEXP \$14, \$1, \$13 // exp(tmp) VAS \$15, \$1, \$14, #1 // 1+exp(tmp) VDV \$16, \$1, \$14, \$15 // y=exp(tmp)/(1+exp(tmp)) RV \$17, \$1 // r[i] = random(0,1) VGT \$8, \$1, \$17, \$16 // if(r[i]>y[i])? VSTORE \$8, \$1, #500 // store hidden vector to address (500)</pre>
---	--	---

Figure 3. 8 Cambricon program fragments of MLP, pooling, and BM [93]

In article [94], the open-source standard instruction sets (RISC-V) were extended to support basic algorithms, namely convolution, pooling, RELU, and matrix addition. The RISC-V architecture has a significant advantage in extensibility due to its four reserved 64/32-bit instruction coding spaces for arbitrary domain-specific architectures, as shown in Figure 3.9. The domains *custom0* and *custom1* are specifically extensible instruction spaces for arbitrary custom instructions. The remaining two domains (i.e., *custom-2/rv128* and *custom-3/rv128*) are specifically for custom extensible instructions of *rv128*.

Inst[4:2]	000	001	010	011	100	101	110	111
Inst[6:5]								(>32b)
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	≥80b

Figure 3. 9 RISC-V architecture instruction space [94]

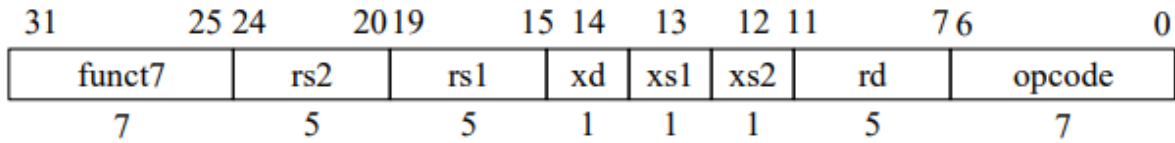


Figure 3. 10 E203 coprocessor extended instruction encoding format [94]

RISC-V processor core has 32-bit with 2-level pipeline known E203 core is used to experiment the extended instruction set, as shown in Figure 3.10 the format of instruction. The format has two source operands (rs1 and rs2) and one destination called rd. The read and write operation is managed by the register index xs1, xs2, and xd. The encoding opcode of the extension spaces is managed by the 0th to 6th bit of the instruction, which is defined as the opcode encoding segment. The performance of machine learning has improved with better speed in the overall result of the article [94].

In the paper [95], an embedded programmable processor for artificial neural networks with its own instruction set architecture was designed. The designed processor is flexible and scalable for artificial neural network workloads with better efficiency. The architecture is based on an Application-Specific Integrated Circuit (ASIC) implementation with 262,000 interconnected, 512 ANs, 32-nm CMOS technology, a silicon area of 2.23 mm², and power consumption of 1.25 mW when operating at 74 MHz and 1.6 V supply. Figure 3.11 shows the instruction format of the

designed ANN processor, while the RNN algorithms were also implemented for testing and analysis within the ANN processor, as shown in Figure 3.12.

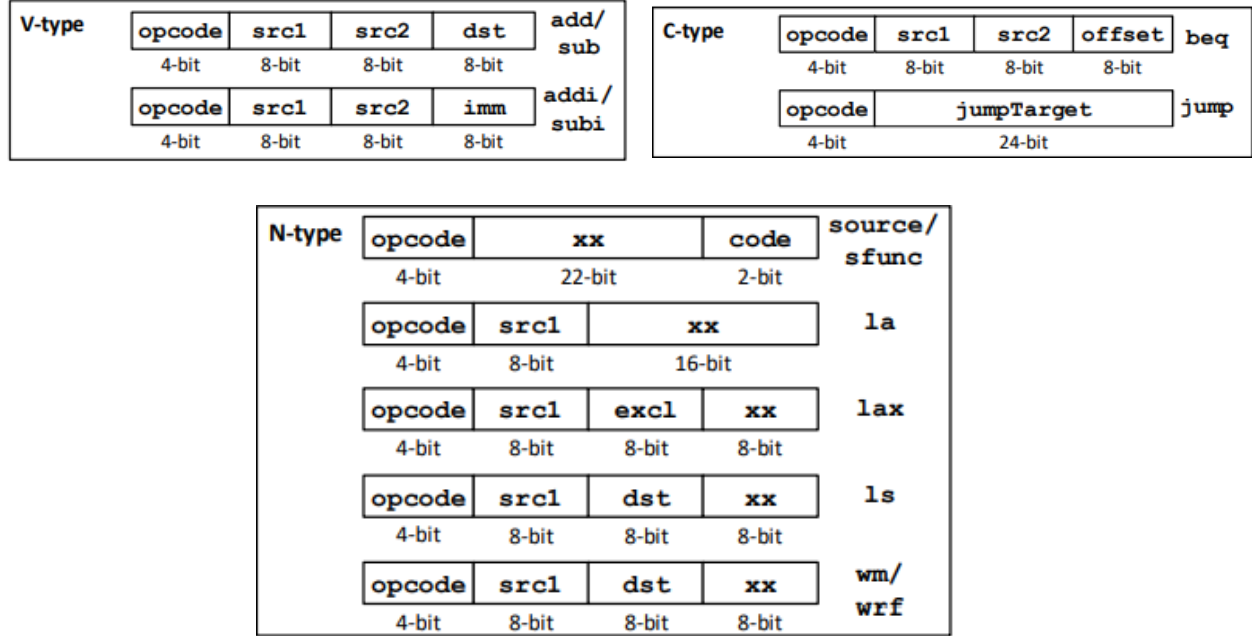


Figure 3. 11 Instruction types and their formats for the designed ANN processor [95]

```

addi $0, $a0, 3      #0 -> 3 time steps
add $0, $0, $a1      #time step counter
addi $0, $c0, 1      #increment var.
sfunc 2              #use ReLU
addi $0, $b0, 10     #output neuron
addi $0, $b1, 9      #HL size
rnnStart:
source 0              #source is input mem.
add $0, $0, $b2       #HL counter
HLWL:
lw $b2, $b2          #WeightMem.b2 -> LPUBank.b2
beq $b2, $b1, loadIMem
add $c0, $b2, $b2     #inc. b2
jump HLWL
loadIMem:
la $a1               #InMem.a1 -> LPUBank.all
add $0, $0, $b2       #HL counter
beq $a1, $0, writeHL
source 1              #Prev. HL outputs
loadPrevHL:
la $b2               #ANMem.b2 -> LPUBank.all
beq $b2, $b1, zeroHL
add $c0, $b2, $b2     #inc. b2
jump loadPrevHL

zeroHL:
add $0, $0, $b2       #HL counter
writeHL:
wm $b2, $b2          #LPUBank.b2 -> ANMem.b2
beq $b2, $b1, OLCheck
add $c0, $b2, $b2     #inc. b2
jump writeHL
OLCheck:
beq $a1, $a0, OL
add $c0, $a1, $a1     #inc. a1
jump rnnStart
OL:
add $0, $0, $b2       #HL counter
lw $b0, $0           #WeightMem.b0 -> LPUBank.0
OLInputs:
ls $b2, $0           #ANMem.b2 -> LPUBank.0
beq $b2, $b1, writeOL
add $c0, $b2, $b2     #inc. b2
jump OLInputs
writeOL:
wm $0, $b0          #LPUBank.0 -> ANMem.b0

```

Figure 3. 12 An example RNN program using our custom-developed ISA [95]

In summary, the chapter described a literature review of general-purpose architecture and hardware accelerators that have been proposed for the execution of deep learning (DL) and neural network (NN) algorithms. The chapter briefly discussed traditional architecture that uses CPU and GPU, FPGA, ASIC, LeNet, AlexNet, ZfNet, GoogLeNet, Cambricon, and accelerators based on RISC-V architecture. The chapter highlights the strengths and limitations of each architecture to identify research gaps that are addressed in this study. Therefore, an overview of machine learning architectures is provided including the machine learning instruction set.

CHAPTER 4

THE PROPOSED INSTRUCTION SET ARCHITECTURE

This chapter introduces the Software M-architecture instruction set. These instructions inherit data-level parallelism and include both one-dimensional array and two-dimensional arrays. Using such kind of ISA will lead to significant advantages in terms of code size. In addition to that, it consequently improves performance. An overview of SPM-architecture's ISA is presented in Section 4.1. The length register is given in Section 4.2. Section 4.3 presents the instruction format. The types of instructions that M-architecture provides are demonstrated in Section 4.5.

4.1 Overview of the M-Architecture ISA

The M-architecture defines a software programmed local memory. The local memory is divided into **sixteen memory regions**, named **M0** to **M15** (or **m0** to **m15**), that can be addressed directly by the memory instructions. A memory region has **MRL bytes**, where **MRL** is a hardware parameter that varies with implementation. A memory region can store integer and floating-point data of various sizes.

The memory region length (**MRL**) is **identical** for **all** memory regions. For example, if **MRL** is **2^{19} bytes** then the total size of all sixteen memory regions is $512 \text{ KiB} \times 16 = 8 \text{ MB}$.

Region M0	Region M1	. . .	Region M15
MRL	MRL	MRL	MRL
Bytes	Bytes	Bytes	Bytes

Figure 4. 1 Software Programmed Local Memory

Figure 4.2 shows each memory region has a corresponding **16-bit length register** that holds the **count of bytes** that are stored inside the region. The length registers of the memory regions are named **L0** to **L15**.

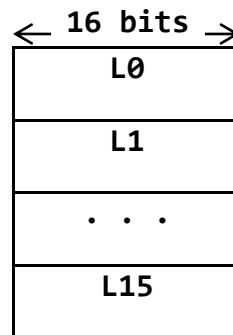


Figure 4. 2 Length Registers

When a program starts execution, the length registers **L0** to **L15** are **zeroed**, indicating that the memory regions are **initially free** (or **empty**). An instruction that loads data or writes a memory region changes its length register.

The M-architecture defines instructions that transfer data directly between global memory and the local memory regions. It also defines computation instructions that operate directly on the memory regions.

4.2 General-Purpose Registers (GPRs)

There are **thirty-two** general-purpose registers (GPRs), named **R0** to **R31**. **R0** is **hardwired to zero**. Reading **R0** returns the constant zero. Writing **R0** has no effect. The result value is discarded. All general-purpose registers are 64-bit wide. They can store integers, addresses, and floating-point data. Using general-purpose registers for both integer and floating-point data simplifies the architecture. It eliminates the need to copy registers between two register files when only one is

used. The same general-purpose registers are also used to pass integer and floating-point parameters to a function, which simplifies the function call convention.

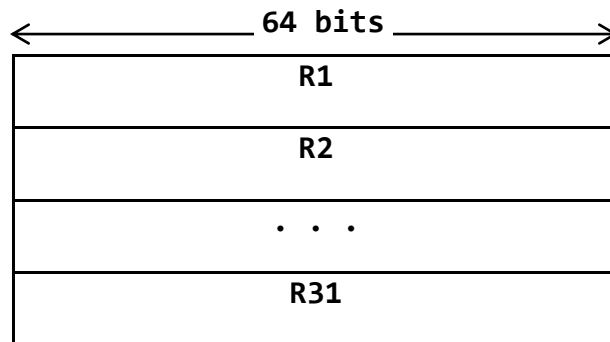


Figure 4. 3 General-Purpose registers

The function call standard defines the role and usage of these registers in functions. Registers R1 to R9 are used to pass parameters to a function. Up to nine parameters (integer and float) can be passed in registers R1 to R9. If there are more than nine parameters, then the additional ones should be passed in memory region m15. Registers R1 to R9 can be used also as temporary registers when there are few parameters. Multiple results can be returned in registers R10, R11, etc. This is usually restricted to few results according to the high-level programming language. If a callee function wants to return a large structure (or object), then it can store its local or global memory and returns its address in R10. Registers R10 to R19 are also used as temporary registers. They are given special names by the assembler: T0 to T9 or t0 to t9. Registers R1 to R19 can be modified freely by a function without saving their values. However, if a caller function needs to preserve some of them, then it is the responsibility of the caller to save their values before making a function call. Registers R20 to R30 are the callee-saved registers. Any function that wants to use these registers must first save their values before modifying them and restore their original values before returning to the caller. Memory region M15 is used as the default stack for preserving registers. If M15 becomes full then it should be saved in global memory. Registers R20 to R29 are given special

names S0 to S9. Register R30 is the Stack Pointer SP that points to the top of a stack segment in global memory. Register R31 is the Link Register LR that saves the return address of a function. It must be saved by a caller function before making a function call and restored before making a function return

Register	Special Name	Role in the function call standard	Saver
R0		Hardwired to Zero	
R1 to R9		Function Parameters / Temporary registers	Caller
R10 to R19	T0 to T9	Function Results / Temporary registers	Caller
R20 to R29	S0 to S9	Saved registers	Callee
R30	SP	Stack Pointer	Callee
R31	LR	Link Register	Caller

Figure 4. 4 Register Usage and the Function Call Standard

4.3 Program Counter Register

The Program Counter (PC) register stores an instruction address. All instructions are 4-byte long and aligned in memory. The least significant two bits of the address are always zero. However, these two zero bits are implicit. They are not stored inside the PC register. If PC is the address of the current instruction, then (PC+1) is the address of the next instruction in memory. For example, the JAL (Jump and Link) instruction saves (PC+1) in R31.

The length of the **PC** register is implementation-specific. A given implementation might limit the number of address bits to reduce the address space.

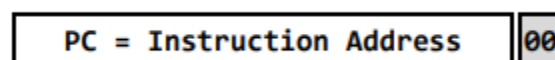


Figure 4. 5 Program Counter Register

4.4 Instruction Set Formats

ALL instructions are 32-bit long and aligned in memory. M-Architecture has four main instruction formats: The R-Format is used mainly by Data Transfer and Computation instructions. The I-Format is used mainly by ALU immediate instructions. The B-Format is used mainly by Branch and Indirect Jump instructions. The J-Format is used mainly by Jump instructions as shown in Figure 4.6.

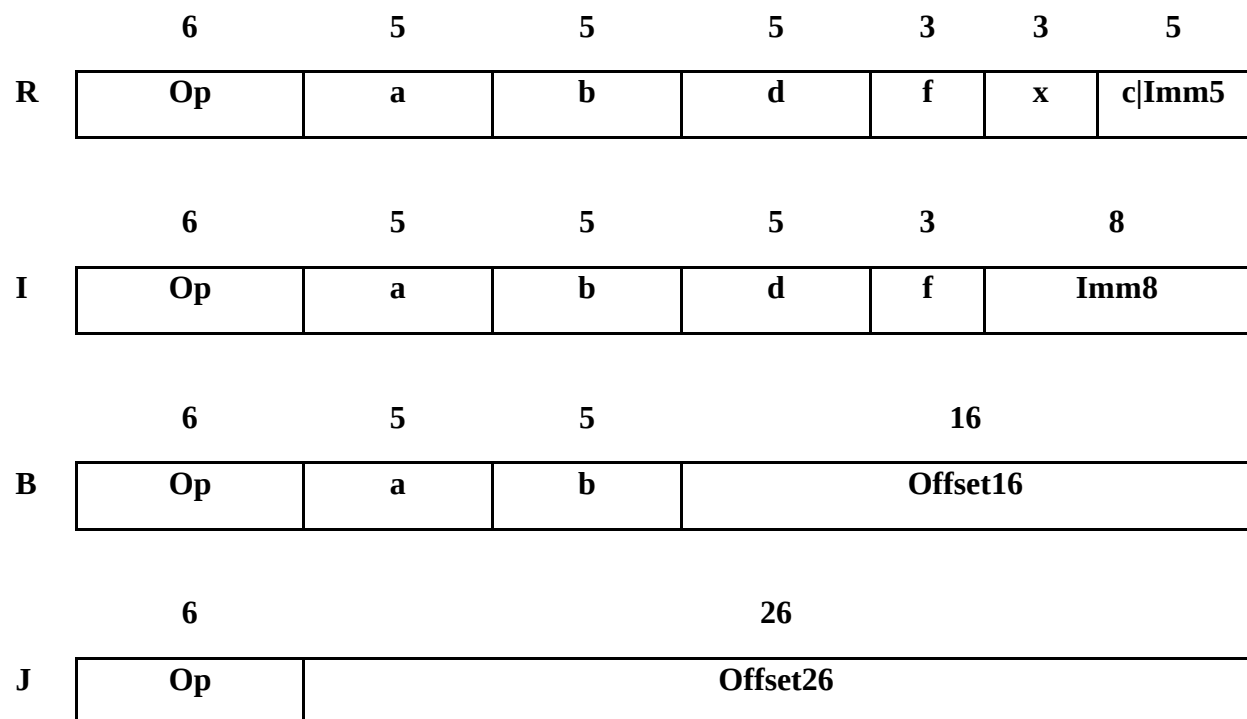


Figure 4. 6 Instruction Set Formats

Here is a short description of the instruction fields:

- Op:** 6-bit opcode, used in all formats
- a:** 5-bit number of source/destination register Ra
- b:** 5-bit number of source register Rb
- c:** 5-bit number of source register Rc (R-format only)

d: 5-bit number of destination register Rd (R-format and I-format)

f: 3-bit function code (R-format and I-format)

x: 3-bit function extension (R-format only)

Imm5: 5-bit immediate (R-format only)

Imm8: 8-bit immediate (I-format only)

Offset16: 16-bit signed offset (B-format only)

Offset26: 26-bit signed offset (J-format only)

Ra, Rb, Rc, and Rd are sometimes replaced by memory regions in the R-format and I-format:

Ma: Source memory region used as a mask for conditional memory computation instructions

Mb: Source memory region for memory computation and read instructions

Md: Destination memory region for memory computation and write instructions

Mc: Source memory region for store memory and destination for load memory instructions

4.5 Types of Instructions

The M-architecture contains eight types of instruction which are *Load and Store Instruction*, *Read and Write instructions*, *Scalar ALU instructions*, *Scalar FPU instructions*, *Control flow instructions*, *Integer Memory computation instructions*, *Integer Memory computation*, *Floating-Point Memory computation*, and *Memory reduction instructions*. More details of the type of each instruction are described in the following sections.

4.5.1 Scalar ALU Instructions

Five opcodes define the scalar ALU instructions: **ALU**, **ALUI**, **ALUJ**, **ALUK**, and **ALUT**. The R-format ALU opcode with $x = 0$ to 2 defines instructions that read two source registers (**Rb** and **Rc**) and write two destination registers (**Ra** and **Rd**). The rationale of writing two registers is that two 64-bit results or a large 128-bit result can be computed. The syntax of an ALU instruction with two source and two destination registers is shown below, where **F** is the instruction mnemonic and function.

F Ra, Rd = Rb, Rc # ALU with two destination registers

The I-format **ALUI**, **ALUJ**, and **ALUK** opcodes define instructions that read **one source register Rb**, use an immediate **Imm** as a second operand, and write **two destination registers: Ra** and **Rd**.

F Ra, Rd = Rb, Imm # ALUI, ALUJ, and ALUK instructions

The R-format **ALU** opcode also defines instructions that read **three source registers (Ra, Rb, and Rc)** and write one destination register **Rd**. The rationale of having three source operands, rather than limiting them to two, is that two dependent operations can be performed by a single instruction. The syntax of an **ALU** instruction with three source registers is shown below. Notice that **Ra** is used as a third operand.

F Rd = Rb, Rc, Ra # ALU with three source registers

The I-format **ALUT** opcode defines instructions that use **three operands (Ra, Rb, and Imm)** and write one destination register **Rd**. Notice that **Imm** replaces **Rc** and **Ra** is used as a third operand.

F Rd = Rb, Imm, Ra # ALUT instructions with three operands

	6	5	5	5	3	3	5
R	Op = ALU	Ra	Rb	Rd	f	x	Rc
I	Op = ALUI	Ra	Rb	Rd	f	Imm8	
I	Op = ALUJ	Ra	Rb	Rd	f	Imm8	
I	Op = ALUK	Ra	Rb	Rd	f	Imm8	
I	Op = ALUT	Ra	Rb	Rd	f	Imm8	

Figure 4. 7 Instruction Formats For Scalar ALU

To enhance readability, the same mnemonics are used in the R-format and I-format ALU instructions. The assembler translates the opcode properly based on the instruction syntax. The instruction syntax is not case-sensitive. The instruction mnemonic and the register names can be written in upper or lower case.

4.5.2 Scalar FPU Instructions

Scalar FPU instructions operate on scalar registers and compute a scalar floating-point result. One R-format FPU opcode defines the scalar floating-point instructions that operate on double-precision 8-byte elements. Floating-point instructions can read at most three source registers (Ra, Rb, and Rc) and write one destination register Rd. The f and x fields allow the definition of at most 64 functions.

	6	5	5	5	3	3	5
R	Op = FPU	Ra	Rb	Rd	f	x	Rc

Figure 4. 8 Format for Scalar FPU Instruction

The syntax of an FPU instruction with one, two, and three source registers is shown below, where F is the instruction mnemonic and function. The instruction syntax is not case-sensitive. The instruction mnemonic and the register names can be written in upper or lower case.

F Rd = Ra // FPU with one source register

F Rd = Ra, Rb // FPU with two source registers

F Rd = Ra, Rb, Rc // FPU with three source registers

The FPU opcode defines eighteen function codes that are divided into three groups:

Functions with x = 0 to 2 are binary and use two source registers: Rb and Rc (Ra is encoded as R0). Functions with x = 3 and 7 are ternary and use three source registers: Ra, Rb, and Rc. Functions with x = 4 to 6 are unary and use source register Rb (Ra and Rc are encoded as R0).

4.5.3 Control Flow Instructions

Control flow instructions modify the program counter register. These are jump and conditional branch instructions. Control flow instructions modify the Program Counter (PC) register. Some instructions also modify a general-purpose register as a side effect. Twelve control flow instructions are defined, each having a unique opcode. The following is the sample control instructions conditional Jump Instructions (J-Format):

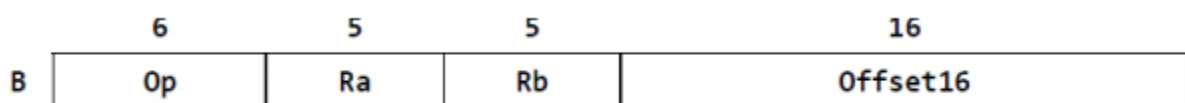


Figure 4. 9 Branch Instruction Format

Ra: Name (and value) of source/destination register

Rb: Name (and value) of a source register

BEQ and BNE compare the bits of any data type

BLT and BGE compare signed integers

BLTU and BGEU compare unsigned integers

IBNE and DBNE increment/decrement Ra as a side effect

Offset16: 16-bit PC-relative signed offset for position-independent branch. Branch Target Address = PC + sign_extend(Offset16). The following is powerful branch instruction for neural networks.

BNES Ra, Rb, @label // PC += (Ra != Rb)? Offset16: 1; Ra-=Rb

Unconditional Jump Instructions (J-Format):

	6	26
J	Op = J	Offset26
J	Op = JAL	Offset26

Figure 4. 10 Unconditional Branch Instruction Format

J @label // PC = PC + Offset26

JAL @label // (R31, PC) = (PC+1, PC+Offset26)

The next branch instruction is JALR Instruction with the following format in Figure 4.11. The JALR (Jump-and-Link-Register) instruction does two things in parallel: 1. Saves the address of next instruction in a destination register: Ra = PC+1 2. Jumps to register-displacement address: PC = Rb + sign_extend(Imm16)

	6	5	5	16
B	Op = JALR	Ra	Rb	Imm16

Figure 4. 11 Jump-and-Link Register Instruction Format

Assembly Language Syntax:

JALR Ra = Rb, Imm16 // (Ra, PC) = (PC+1, Rb+Imm16)

JALR Ra = Rb // Pseudo: JALR Ra = Rb, 0

Description:

Ra: Name (and value) of destination register

Rb: Name (and value) of source register, and Imm16: 16-bit signed Immediate.

4.5.4 Load and Store Memory Instructions

Load and Store memory instructions transfer data directly between global memory and the local memory regions. Two R-format opcodes, **LM** and **SM**, define all memory-to-memory transfer instructions.

	6	5	5	5	3	3	5
R	Op = LM	Ra	Rb	Rd	f	x	Mc
R	Op = SM	Ra	Rb	Rd	f	x	Mc

Figure 4. 12 Load Instruction Format

The **LM** opcode defines the **load memory** instructions that transfer a contiguous block of bytes from global memory into the target memory region **Mc**. The **LM** opcode with **x = 0** to **3** defines four functions. The **LM** function does extend bytes, while **LM1D**, **LM2D**, and **LM4D zero-extend** each **1-byte**, **2-byte**, and **4-byte** element, respectively, to double the block size in the target memory region **Mc**. The **LM** opcode with **x = 4** to **5** defines the **append memory** functions that transfer a block of bytes from global memory and append it into the target memory region **Mc**

starting at **index Lc**. Data is transferred and appended if there is free space in **Mc**. Four function codes are defined: the **AM** function does not extend the bytes, while the **AM1D**, **AM2D**, and **AM4D** functions **zero-extend** each **byte**, **2-byte**, and **4-byte** element, respectively, to double the block size in the target memory region **Mc**.

Op = LM	f = 3-bit function codes for the LM opcode			
x = 0 to 3	LM = 0	LM1D = 1	LM2D = 2	LM4D = 3
x = 4 to 5	AM = 0	AM1D = 1	AM2D = 2	AM4D = 3

Figure 4. 13 Instructions for LM Opcode

The **SM** opcode defines the **store memory** instructions that transfer data for a source memory region **Mc** into global memory. Four functions are defined: the **SM** function transfers a block of bytes, while **SM1L**, **SM2L**, and **SM4L** transfer the **low 1-byte**, **2-byte**, and **4-byte** of each element to reduce the block size by half when transferred into global memory.

Op = SM	f = 3-bit function codes for the SM opcode			
x = 0 or 1	SM = 0	SM1L = 1	SM2L = 2	SM4L = 3

Figure 4. 14 Instructions for SM Opcode

4.5.5 Read and Write Instructions

Read and write instructions transfer data between a local memory region and general-purpose registers. Two R-format opcodes, RD and WR, define all the read and write instructions. The RD and WR instruction R-format uses a small 5-bit immediate instead of register Rc. Figure 4.17 shows the format of the instructions.

	6	5	5	5	3	3	5
R	Op = RD	Ra	Mb	Rd	f	x	Imm5
R	Op = WR	Ra	Rb	Md	f	x	Imm5

Figure 4. 15 The Read and Write Instructions Format

The **RD** opcode transfers data from source memory region Mb into destination register Rd. The RDn (Read) functions transfer n = 1, 2, 4, or 8 bytes and zero-extend the data in Rd. The RSn (Read Signed) functions transfer n = 1, 2, or 4 bytes and sign-extend the data in Rd. The Ra register specifies the index, which is scaled by n, to address memory region Mb. The **WR** opcode transfers data of a source register Rb into destination memory region Md. The WRn (Write) functions write the lower n = 1, 2, 4, or 8 bytes of Rb into memory region Md. The Ra register specifies the index, which is scaled by n, to address memory region Md. The **PUSH** instruction writes multiple registers at the end of memory region Md, used as a stack, and updates its length register Ld. The **POP** instruction reads multiple registers at the top of source memory region Mb, used as a stack, and updates its length register Lb.

4.5.5.1 Reading a Memory Region

Reading a memory region instruction is under opcode of RD (with x = 0) with seven functions which are RD1, RD2, RD4, RD8, RS1, RS2, RS4. Figure 4.17 shows the format of reading instructions where Mb is the local memory region to read, Ra is a name (and value) of the local index register, Rd is a name (and value) of the destination register, and Imm5 5-bit signed immediate (-16 to +15). The RDn functions transfer n = 1, 2, 4, or 8 bytes from source memory region Mb and zero-extend the data when written into destination register Rd. The RSn functions transfer n = 1, 2, or 4 bytes from source memory region Mb and sign-extend the data when written

into destination register Rd. Register Ra specifies the local index. The RDn and RSn function scales the Ra value by n to compute the local memory address and ensure alignment.

Assembly-Language Syntax:

RD1 Rd = Mb[Ra] , Imm5 // Rd=RD1(&Mb[Ra*1]); Ra+=Imm

RD2 Rd = Mb[Ra] , Imm5 // Rd=RD2(&Mb[Ra*2]); Ra+=Imm

RD4 Rd = Mb[Ra] , Imm5 // Rd=RD4(&Mb[Ra*4]); Ra+=Imm

RD8 Rd = Mb[Ra] , Imm5 // Rd=RD8(&Mb[Ra*8]); Ra+=Imm

RS1 Rd = Mb[Ra] , Imm5 // Rd=RD1(&Mb[Ra*1]); Ra+=Imm

RS2 Rd = Mb[Ra] , Imm5 // Rd=RD2(&Mb[Ra*2]); Ra+=Imm

RS4 Rd = Mb[Ra] , Imm5 // Rd=RD4(&Mb[Ra*4]); Ra+=Imm Mb Write a Memory

Register Ra is post-updated: Ra += Imm5. If Imm5 is not specified, it defaults to 0 and Ra is unmodified.

RDn Rd = Mb[Ra] // Pseudo: RDn Rd = Mb[Ra], 0

RSn Rd = Mb[Ra] // Pseudo: RSn Rd = Mb[Ra], 0

If the Ra value is outside the indexing range, then the Read exception is raised: if $GEU(Ra, Lb*8/n)$ Exception(Read, PC, Ra) Little-Endian byte ordering is used by the RDn and RSn functions ($n>1$).

4.5.5.2 Writing a Memory Region

Writing a Memory Region instructions is transfer byte elements from the sources register to the local memory. WR is the opcode of the instructions with four functions which are WR1, WR2, WR4, WR8 as shown in Figure 4.17 the format of WR opcode.

Assembly-Language Syntax:

WR1 Md[Ra] , Imm5 = Rb // WR1(&Md[Ra*1],Rb); Ra+=Imm5

WR2 Md[Ra] , Imm5 = Rb // WR2(&Md[Ra*2],Rb); Ra+=Imm5

WR4 Md[Ra] , Imm5 = Rb // WR4(&Md[Ra*4],Rb); Ra+=Imm5

WR8 Md[Ra] , Imm5 = Rb // WR8(&Md[Ra*8],Rb); Ra+=Imm5

Md is the local memory region to write, Ra is the name (and value) of local index register, Rb is the name (and value) of source data register, Imm5 5-bit signed immediate (-16 to +15).

The WRn functions transfer the lower $n = 1, 2, 4, \text{ or } 8$ bytes of a source register Rb into local destination memory region Md. Register Ra specifies the local index. The WRn functions scale the Ra value by n to compute the local memory address and ensure alignment. Register Ra is post-updated: $Ra += Imm5$. If Imm5 is not specified, it defaults to 0 and Ra is unmodified. $WRn\ Md[Ra] = Rb$ // Pseudo: $WRn\ Md[Ra], 0 = Rb$ If the Ra value is outside the indexing range, then the Write exception is raised: if $GEU(Ra, Ld*8/n)$ Exception(Write, PC, Ra) Little-Endian byte ordering is used by the WRn functions ($n > 1$).

4.5.5.3 Push Instruction

The PUSH instruction writes (n+1) doublewords at the end of a local memory region Md, which is used as a stack. It transfers the value of source register Ra plus n additional values starting at register Rb. The data is written starting at local index Ld, which is the length register of Md.

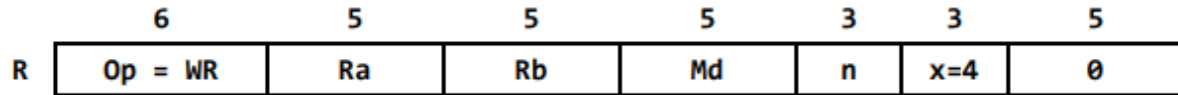


Figure 4. 16 Push Instruction Format

Figure 4.18 shows the format of PUSH instruction under WR opcode with function f is equal to the number of additional registers to push (n). Assembly-Language Syntax as follows:

PUSH Md = Ra, Rb, n // Push Ra, Rb, ... onto Md

Md is the local memory region to write as a stack, Ra is the name of the first source register, Rb is the name of a sequence of additional source registers, n is the number of additional source registers to push (n = 0 to 7), and the Rc is the field is not used and should be encoded as zero.

If n is 0, then Ra is the only source register. If n is 7 then Ra, Rb, R[b+1] ... R[b+6] are eight source registers. At most eight double words (64 bytes) can be pushed.

Example: PUSH m15 = r31, r20, 2 // Push r31, r20, r21 onto m15

If there is no sufficient space in Md then the PushOverflow exception is raised. The exception is handled by storing Md in global memory using the stack pointer register SP = R30 to free space in Md. The (n+1) register values are then stored in Md and the length register Ld is post-updated.

```
if (Ld+n+1 > MRL) Exception(PushOverflow, PC, Md);
```

```
WR8 Md[Ld] = Ra; WR8 Md[Ld+1] = Rb; ...
```

```
Ld += (n+1);
```

The PUSH instruction syntax can be simplified, when one or two source registers are used:

```
PUSH Md = Ra // Pseudo: PUSH Md = Ra, R0, 0
```

```
PUSH Md = Ra, Rb // Pseudo: PUSH Md = Ra, Rb, 1
```

4.5.5.4 Pop Instruction

The POP instruction reads (n+1) doublewords at the top of a local memory region Mb, which is used as a stack. It transfers these values into destination register Ra plus n additional registers starting at Rd.

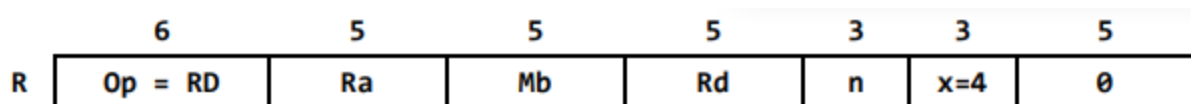


Figure 4. 17 Pop Instruction Format

Figure 4.19 shows the format of POP instruction under RD opcode with function, f is equal to the number of additional registers to pop (n). Assembly-Language Syntax as follows:

```
POP Ra, Rd, n = Mb // Pop Ra, Rd, ... from Mb
```

Mb is the local memory region to read as a stack, Ra is the name of the first destination register.

Rd is the name of a sequence of additional destination registers, n is the number of additional destination registers to pop (n = 0 to 7), and the Rc field is not used and should be encoded as zero.

The length register Lb of Mb is pre-decremented before reading the n double words. If Lb becomes negative, then the PopUnderflow exception is raised. The exception is handled by loading Mb from global memory using the stack pointer register SP = R30.

```
Lb -= (n+1);
```

```
if (Lb < 0) Exception(PopUnderflow, PC, Mb);
```

```
RD8 Ra = Mb[Lb]; RD8 Rd = Mb[Lc+1]; ...
```

The POP instruction syntax can be simplified, when one or two destination registers are used:

```
POP Ra = Mb // Pseudo: POP Ra, R0, 0 = Mb
```

```
POP Ra, Rd = Mb // Pseudo: POP Ra, Rd, 1 = Mb
```

4.5.6 Integer Memory Computation Instructions

Integer memory instructions perform integer computation on memory regions. They read and write memory regions. These instructions can be conditional and use a memory region as a mask. Four opcodes define the integer memory computation instructions: ALUMM8, ALUMR8, ALUMM4, and ALUMR4. The ALUMM8 and ALUMM4 opcodes define functions that read three source memory regions Ma, Mb, Mc, and write destination memory region Md. The ALUMR8 and ALUMR4 opcodes define functions that read two source memory regions Ma and Mb, a scalar register Rc, and write destination region Md.

The ALUMM8 and ALUMR8 opcodes define functions that operate on 8-byte integer elements. The ALUMM4 and ALUMR4 opcodes define functions that operate on 4-byte integer elements. The f and x fields allow the definition of at most 64 memory computation functions per opcode.

The architecture can be extended to include opcodes that operate on 2-byte and 1-byte integer elements.

	6	1	4	5	5	3	3	5
R	Op = ALUMM8	!	Ma	Mb	Md	f	x	Mc
R	Op = ALUMR8	!	Ma	Mb	Md	f	x	Rc
R	Op = ALUMM4	!	Ma	Mb	Md	f	x	Mc
R	Op = ALUMR4	!	Ma	Mb	Md	f	x	Rc

Figure 4. 18 Formats for Integer Memory Computation Instructions

The syntax of a conditional integer memory computation instruction is shown in Figure 4.20, where Fn is the instruction mnemonic and function, Ma is an optional memory mask with optional! operator, Md is a destination memory region, Mb is a first source memory operand, and Mc|Rc is a second memory operand or scalar register. The length of destination memory Md is Ld=Lb, which is the length of Mb.

Fn (!Ma) Md = Mb, Mc|Rc // if (!Ma_i) Md_i = Fn (Mb_i, Mc_i|Rc)

The optional memory mask Ma, enclosed in parentheses, is restricted to M0 to M14. If the memory computation instruction is unconditional, then the memory mask does not appear in the instruction syntax and Ma is encoded as 15 in the instruction format. The ! (not bit) that appears before Ma indicates whether zero Boolean values should enable the writing of elements in Md.

The second group of memory computation instructions is unconditional and uses Ma as a third operand. The syntax is shown below, where Fn is the instruction mnemonic and function, Md is the destination memory, Mb is the first memory operand, Mc or Rc is the second operand, and Ma is the third operand. The length of the destination memory region Md is Ld=Lb, which is the length of Mb.

Fn Md = Mb, Mc|Rc, Ma

To enhance readability, the same mnemonic (function Fn) is used in the ALUMMn and ALUMRn opcodes. The assembler translates the opcode properly (ALUMMn or ALUMRn) based on the instruction syntax.

The instruction syntax is not case-sensitive. The instruction mnemonic (function Fn), memory regions, and register names can be written in upper or lower case. There are Addition Instructions, Compare Instructions, Memory logic Instructions, Memory Shift Instructions, Memory-Multiple and Divide Instructions, and three operand integer memory computation instructions.

4.5.7 Floating-Point Memory

Floating-Point Memory computation instructions perform floating-point computation on memory regions. These instructions can be conditional and use a memory region as a mask. Four opcodes define the integer memory computation instructions: FPUMM8, FPUMR8, FPUMM4, and FPUMR4. The FPUMM8 and FPUMM4 opcodes define functions that read three source memory regions Ma, Mb, Mc, and write destination memory region Md. The FPUMR8 and FPUMR4 opcodes define functions that read two source memory regions Ma and Mb, a scalar register Rc, and write destination region Md.

	6	1	4	5	5	3	3	5
R	Op = FPUMM8	!	Ma	Mb	Md	f	x	Mc
R	Op = FPUMR8	!	Ma	Mb	Md	f	x	Rc
R	Op = FPUMM4	!	Ma	Mb	Md	f	x	Mc
R	Op = FPUMR4	!	Ma	Mb	Md	f	x	Rc

Figure 4. 19 Format for Floating-Point Memory Computation Instructions

The FPUMM8 and FPUMR8 opcodes define functions that operate on 8-byte floating-point elements. The FPUMM4 and FPUMR4 opcodes define functions that operate on 4-byte floating-point elements. The f and x fields allow the definition of at most 64 memory computation functions per opcode.

The syntax of a conditional floating-point memory computation instruction is shown below, where Fn is the instruction mnemonic and function, Ma is an optional memory mask with optional! operator, Md is a destination memory region, Mb is a first source memory operand, and Mc|Rc is a second memory operand or scalar register. Some floating-point memory computation functions are unary and do not have a second operand. The length of destination memory Md is Ld=Lb, which is the length of Mb.

Fn (!Ma) Md = Mb // if (!Ma_i) Md_i = Fn (Mb_i) Fn (!Ma)

Md = Mb, Mc|Rc // if (!Ma_i) Md_i = Fn (Mb_i, Mc_i|Rc)

The optional memory mask Ma, enclosed in parentheses, is restricted to M0 to M14. If the memory computation instruction is unconditional, then the memory mask does not appear in the instruction syntax and Ma is encoded as 15 in the instruction format. The ! (not bit) that appears before Ma indicates whether zero Boolean values should enable the writing of elements in Md.

The second group of memory computation instructions is unconditional and uses Ma as a third operand. The syntax is shown below, where Fn is the instruction mnemonic and function, Md is the destination memory, Mb is the first memory operand, Mc or Rc is the second operand, and Ma is the third operand. The length of destination memory Md is Ld=Lb, which is the length of Mb.

Fn Md = Mb, Mc|Rc, Ma

To enhance readability, the same mnemonic (function Fn) is used in the FPUMMn and FPUMRn opcodes. The assembler translates the opcode properly (FPUMMn or FPUMRn) based on the instruction syntax.

The instruction syntax is not case-sensitive. The instruction mnemonic (function Fn), memory regions, and register names can be written in upper or lower case. Floating-point involves the memory addition instructions, memory compare instructions, memory multiple and divide instructions, three operand floating-point memory computation instructions, memory conversion instructions, reciprocal, square root, exponential, log base two, and trigonometrical instructions.

4.5.8 Memory Reductions Instructions

Memory reduction instructions perform computation across the elements of a memory region and compute a scalar result, which is written into a register.

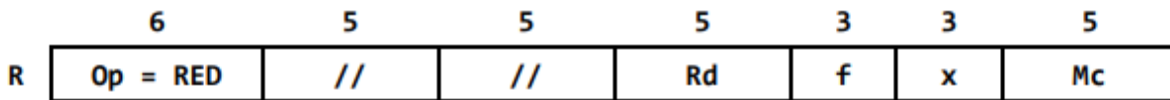


Figure 4. 20 Memory Reduction Instruction Format

Memory reduction functions include adding all the elements of a memory region or computing the minimum or maximum. The following are memory reduction instructions.

RSUM8 Rd = Mc // Reduction SUM (8-byte elements)

RAND8 Rd = Mc // Reduction AND (8-byte elements)

ROR8 Rd = Mc // Reduction OR (8-byte elements)

RXOR8 Rd = Mc // Reduction OR (8-byte elements)

RMIN8 Rd = Mc // Reduction MIN (8-byte elements)

RMAX8 Rd = Mc // Reduction MAX (8-byte elements)

RMINU8 Rd = Mc // Reduction MINU (8-byte elements)

RMAXU8 Rd = Mc // Reduction MAXU (8-byte elements)

RFSUM8 Rd = Mc // Reduction FSUM (8-byte elements)

RFMIN8 Rd = Mc // Reduction FMIN (8-byte elements)

RFMAX8 Rd = Mc // Reduction FMAX (8-byte elements)

4.6 Code example for some of our machine learning operations

This section will demonstrate the beneficial of our instructions into some machine learning operations that can be executed through our general-purpose instructions.

4.6.1 Dot Product

The dot product operation is used in the Convolution layer in CNN between the input and filter to extract feature map output as shown in Equation 2.5. The operation uses RSUM instruction as part of memory reduction instructions. Consider computing the dot product of two integer byte arrays A and B. This can be expressed in vector notation:

$$\mathbf{s} = \text{sum}(\mathbf{A}[0:n] * \mathbf{B}[0:n])$$

The dot product is a scalar value computed by the sum reduction. Consider the given that r1 and r2 are the global addresses of arrays A and B, and r3 is the count of the 1-byte integers elements to reduce, the following loop computes the dot product of A and B and stores its scalar value in register r4.

```

# Dot Product of two floating-point arrays: A and B

# r1 = Address of array A in global memory

# r2 = Address of array B in global memory

# r3 = Count n of 8-byte elements in A and B

mov r4 = r0 # Initialize: r4 = 0

@dotproduct

lm m0, r5 = [r1]+ r3 # L0=..; m0=LM(r1,L0); r1+=L0*1; r5=L0

lm m1 = [r2]+ r3 # L1=..; m1=LM(r2,L1); r2+=L1*1

mul1 m2 = m0, m1 # L2=L0; m2=MUL1(m0,m1,L0)

rsum1 r4 = m2, r4 # R=RSUM1(m2,L2); r4=rSUM1(R,r4)

bnes r3, r5, @dotproduct # branch if (r3!=L0); r3-=L0

```

The mov instruction outside the loop initializes $r4 = 0$. The two lm instructions set $L0 = L1 = \text{MINU}(r3, \text{MRL})$ bytes and load two blocks of arrays A and B into $m0 = \text{LM8}(r1, L0)$ and $m1 = \text{LM8}(r2, L1)$. The address registers r1 and r2 are post-updated. The first lm instruction returns $r5=L0$. The mul1 instruction multiplies the 1-byte integer elements of m0 and m1. The results are written in m2. The length L2 of m2 becomes the length L0 of m0. The rsum1 is a memory reduction instruction that adds the floating-point elements of m2 and reduces them into one double-precision scalar value R, which is added to r4 and then rewritten in r4. The length register L2 specifies the number of elements of m2 to reduce. The bnes instruction repeats the execution of the dot product

loop if arrays A and B are large. At most MRL double floats are loaded and reduced each iteration. During the last iteration, the remainder r3%MRL elements are loaded and their dot product is accumulated in r4.

4.6.2 Vector Matrix Multiplication in a Neural Network

In a neural network, the inputs to a hidden layer can be **unsigned bytes**. The edges of a fully connected hidden layer carry **weights** that can be represented by a matrix of **unsigned bytes** from **0** to **255** and map to **0** to **0.996** when divided by **256**. The neurons at the outputs of a hidden layer compute the weighted average of their inputs, using vector-matrix multiplication that saturates results at **255**. The saturated weighted averages pass through a **ReLU** activation function that reduces all values smaller than **128** (or **0.5**) down to **0**. The output of the activation function is either **0** or in the range **128** to **255**.

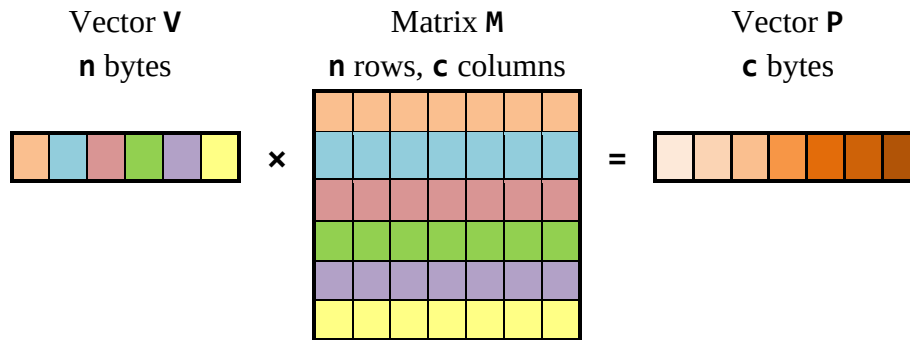


Figure 4. 21 Vector-Matrix Multiplication

The following **VMM** function multiplies an input vector **V** of length **n** bytes by a matrix **M** with **n** rows and **c** columns to produce an output product vector **P** of length **c** bytes that saturate results at **255**. The product vector pass through an activation function that reduces all values below **128** down to **0**. The first **1m** instruction loads vector **V** into **m1**. Vector **V** is assumed to fit in **m1** ($n \leq$

MRL). The **mov** instruction initializes **t2 = r2** = address of matrix **M** and the index register **t0 = 0**.

The **clr1** instruction sets **L3=r5=c** and clears **c** bytes of **m3**, assuming **c ≤ MRL**.

VMM: Vector Matrix Multiply with Rectified Output (Unsigned Bytes)

r1, r2, r3: Global address of V, M, P

r4: Length n (bytes) of the input vector V = Number of rows in M

r5: Length c (bytes) of the output vector P = Number of columns in M

This version assumes that n and c are less than or equal to MRL

@VMM		# VMM Function
lm	m1 = [r1], r4	# L1=n; Load m1 = vector V
mov	t2, t0 = r2, r0	# t2 = address of M, t0 = index i
clr1	m3 = r5	# L3=c; Clear c bytes in m3
@loop		# Loop over the n rows of matrix M
lm	m2 = [t2]+ r5	# L2=c; Load m2 = row M[i]; t2+=c
rd1	t1 = m1[t0], 1	# Read t1=RD1(&m1[t0])=V[i]; t0+=1
muhu1	m4 = m2, t1	# L4=c; m4 = high byte of m2*V[i]
satu1	m3 = m3, m4	# L3=c; m3 = saturated add (m3+m4)
bne	t0, r4, @loop	# branch @loop if (i != n)
min1	m3 = m3, r0	# m3 = Reduce 1 to 127 down to 0
sm	[r3] = m3	# Store vector P = m3
ret		# Return to caller

A loop is written to load each row of matrix **M** into memory region **m2**, multiply this row by one element of vector **V**, and accumulate the product vector in memory region **m3**. The **@loop lm** instruction loads row **i** of matrix **M** from global address **t2** into memory region **m2**. A row of matrix **B** is assumed to fit in a memory region ($c \leq \text{MRL}$). The **[t2]+** option post-updates the address register **t2** to point to the next row in matrix **B**.

The **rd1** instruction reads a **one-byte** element of **m1** at index **t0=i** and transfers its value into **t1**. Register **t0** is also post-updated to index the next byte in **m1**. The **muhu1** (**m**ultiply **h**igh **u**nsigned) instruction multiplies the unsigned byte elements of **m2** (row **i** of **B**) by scalar register **t1=V[i]** and stores the **high-byte** of the product in **m3**. The low-byte of the product is discarded. For example, if an input byte **135** is multiplied by a weight of **54** then the unsigned product is **7290** and the **high byte** of the product is **28**. This is equivalent to multiplying **135** by a weight of **54/256** (equal to **0.21**). The integer product is also **28** and the fraction is discarded. The **sat1** (**s**aturate **u**nsigned) instruction adds the **unsigned bytes** of **m3** and **m4** and saturates the sum at **255**. Saturation is used when a carry is generated, and the unsigned sum is above 255. The **bne** instruction loops back if there are more rows in the weight matrix **M** to multiply-accumulate.

The **min1** instruction, outside the loop, is an activation function with a threshold of **128** (equal to **0.5**). It compares the bytes of **m3** against **0**. If a byte of **m3** is from **1** to **127** then **min1** reduces it to **0**. If a byte is **128** to **255** then **min1** does a **signed** comparison against a negative value from **-128** to **-1** and returns the same value. The output of **min1** is either **0** (if the input is **0** to **127**) or unchanged if **128** to **255**. It implements a **ReLU** activation function with a threshold of **128**. The **sm** instruction, outside the loop, stores the accumulated and rectified product in **m3** in vector **P** at global address **r3**. The **ret** instruction returns back to the caller function.

4.6.3 Vector Matrix Multiply using Signed Weights in a Neural Network

If the inputs to a hidden layer are **signed bytes** that carry positive values from **0** to **+127** and the edges of a hidden layer carry **signed weights** stored in a matrix of **signed bytes** from **-128** to **+127** then the neurons at the outputs of a hidden layer compute the weighted average of their inputs using vector-matrix multiplication that saturates at **-128** and **+127**. The saturated weighted averages pass through a **ReLU** activation function that modifies all negative values to **0**. The output of the activation function is positive values from **0** to **+127**.

The **muh1** (**m**ultiply **h**igh) instruction multiplies the signed byte elements of **m2** (row **i** of **B**) by scalar register **t1=V[i]** and stores the **high-byte** of the product in **m3**. The low-byte of the product is discarded. For example, if an input byte **65** is multiplied by a weight of **-34** then the signed product is **-2210** and the **high byte** of the product is **-9**. This is equivalent to multiplying **65** by weight of **-34/256** (equal to **-0.13**). The integer product using the floor function is **-9**.

The **sat1** (**s**aturate) instruction adds the **signed bytes** of **m3** and **m4** and saturates the sum at either **-128** or **+127**. Saturation is used when there is overflow and the sum out of range. The **max1** instruction, outside the loop, is an activation function with a threshold of **0**. It compares the bytes of **m3** against **0**. If a byte of **m3** is **negative** from **-128** to **-1** then **max1** forces it to become **0**. It implements a **ReLU** activation function with a threshold of **0**.

VMM: Vector Matrix Multiply with Rectified Output (Signed Bytes)

r1, r2, r3: Global address of V, M, P

r4: Length n (bytes) of the input vector V = Number of rows in M

r5: Length c (bytes) of the output vector P = Number of columns in M

This version assumes that n and c are less than or equal to MRL

@VMM		# VMM Function
lm	m1 = [r1], r4	# L1=n; Load m1 = vector V
mov	t2, t0 = r2, r0	# t2 = address of M, t0 = index i
clr1	m3 = r5	# L3=c; Clear c bytes in m3
@loop		# Loop over the n rows of matrix M
lm	m2 = [t2]+ r5	# L2=c; Load m2 = row M[i]; t2+=c
rd1	t1 = m1[t0], 1	# Read t1=RD1(&m1[t0])=V[i]; t0+=1
muh1	m4 = m2, t1	# L4=c; m4 = high byte of m2*V[i]
sat1	m3 = m3, m4	# L3=c; m3 = saturated add (m3+m4)
bne	t0, r4, @loop	# branch @loop if (i != n)
max1	m3 = m3, r0	# m3 = negative bytes are made 0
sm	[r3] = m3	# Store vector P = m3
ret		# Return to caller

In summary, the chapter introduced the proposed M-architecture instruction set. These instructions are efficient and flexible to resolve complex operation blocks into general-purpose instruction corresponding to the low-level operation. These instructions inherit data-level parallelism. Using such kind of ISA will lead to significant advantages in terms of code size. In addition to that, it consequently improves performance.

CHAPTER 5

THE PROPOSED ARCHITECTURE

This chapter gives an overview of the proposed M architecture and describes its main components in detail, presenting some features offered by the target model. The various choices made in the development of our architecture have been pointed out with a brief justification for inappropriate places. In addition, the microarchitecture of the M architecture and the partitioning of memory areas as slices are discussed. Finally, the scalability of the target architecture is presented at the end of the chapter.

5.1 M-architecture Overview

A block diagram of the M-processor is shown in Figure 5.1. There is only an I-cache that fetches instruction blocks from a high-bandwidth global memory. However, there is no data cache. Instead, a large software programmed memory unit is used, which is more efficient in terms of area and energy consumption. The software programmed local memory is under program control and filled only with the needed data. It enables in-memory computation and can be addressed privately without requiring any address translation. On the other hand, data caches require address translation on each access, have a variable access time, can be polluted with unused data, and cannot be used for in-memory computation.

The M-processor pipeline starts by fetching instructions from the I-cache at a given PC address. The fetched instructions wait in an instruction buffer (I-buffer). They can be a mixture of memory transfer, memory computation, scalar, and branch instructions. Next, they are decoded by the I-

decode logic to generate the control signals. Memory instructions are queued in an M-buffer, while scalar instructions are executed by the scalar function units. A memory instruction typically requires reading the length register of a source memory region or updating the length register of a target memory region. The I-decode logic can read and update the length register file (LRegs).

A memory instruction might also require reading one or more scalar registers before starting its execution in the local memory unit. The values of the scalar registers that are needed by the memory instruction are queued in the M-buffer.

Scalar instructions can execute in parallel with memory instructions. They can also complete before the memory instructions, while memory instructions are waiting in the M-buffer. There is no need for many scalar execution units, complex out-of-order execution, or branch prediction. Scalar instruction execution and branch execution can be simplified because the M-processor spends most of its cycles executing memory instructions.

Memory instructions can be either memory transfer or computational. Memory transfer instructions require a high-bandwidth global memory and very wide data busses to support the fast transfer of data between the global memory and the local memory regions. Memory computation instructions operate directly on the local memory regions. Memory transfer and memory computation instructions are overlapped to hide the latency of the global memory access.

Special read and write instructions transfer data between the local memory unit and the scalar registers. Local memory addresses do not require address translation, or a TLB. Address translation is needed only when transferring blocks of data between global memory and the local memory unit.

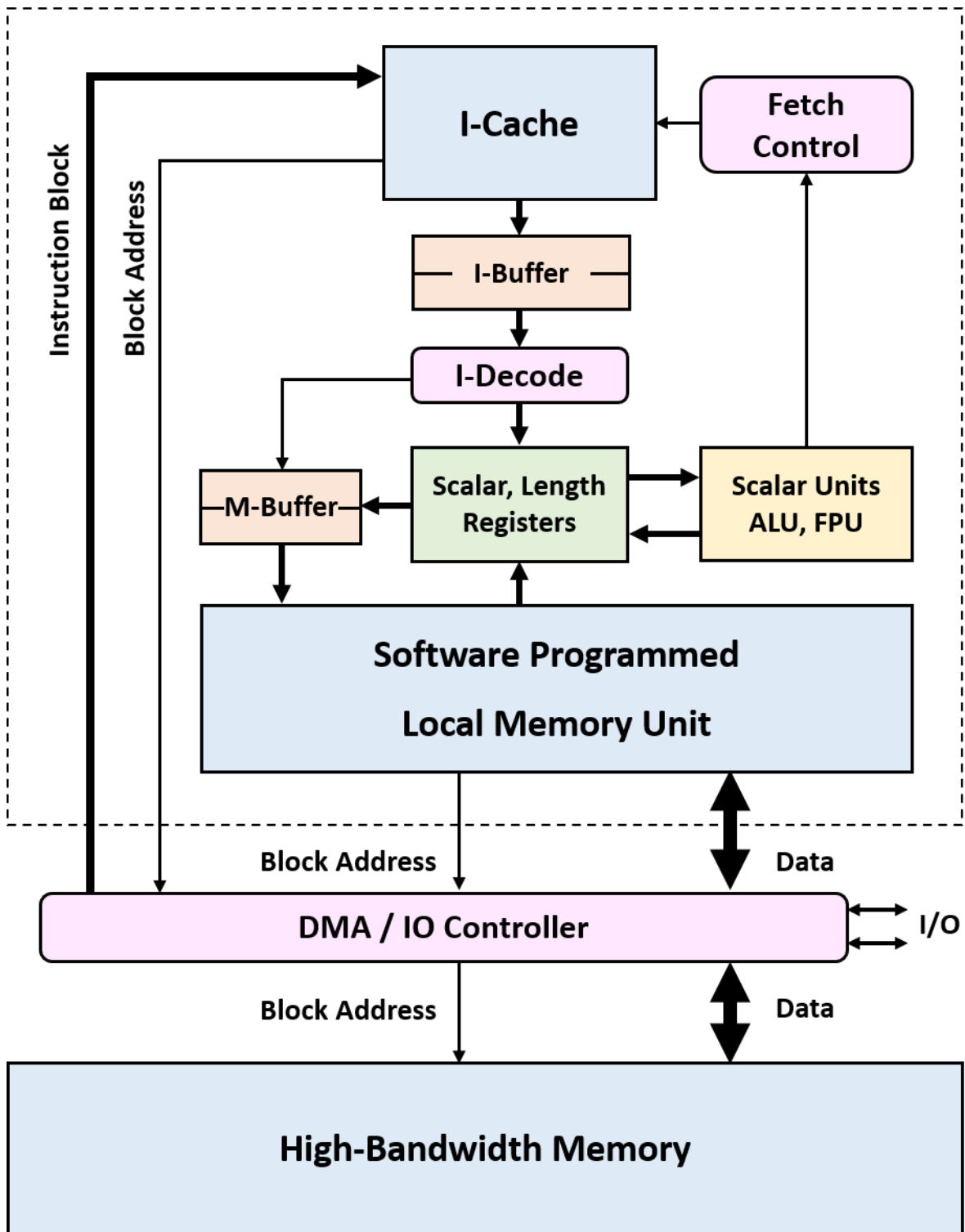


Figure 5. 1 High-Level Block Diagram of the M-Processor

Internally, the software programmed local memory unit is partitioned into many slices (tens or hundreds of slices, depending on implementation). Each slice has its own execution units and is capable of overlapping memory transfer with memory computation.

5.2 Software Programmed Local Memory

A block diagram of the software-programmed memory is shown in Figure 1.7. The local memory regions M0 to M15 are partitioned into NS slices, where NS can be scaled according to implementation. Figure 1.7 shows 16 slices, numbered 0 to 15. Each sliced region has MRL/NS bytes. For example, if $MRL = 2^{15}$ bytes then each sliced region has $2^{15}/2^4 = 2^{11}$ bytes. Low-order block interleaving is used, such that two successive 16-byte blocks are mapped to two adjacent memory slices. Given a 15-bit local address $A[14:0]$ in a memory region, the 4-bit byte offset is $B = A[3:0]$ within a 16-byte block, the 4-bit slice number is $S = A[7:4]$ within the network, and the 7-bit block index is $I = A[14:8]$ within a slice, as shown in Figure 1.6. If the number of slices is scaled from 16 to 64 then we need an increased 6-bit slice field $S = A[9:4]$ and a reduced 5-bit block index field $I = A[14:10]$.



Figure 5. 2 Mapping a Local byte Address, A onto a Slice, S, and Index I

Each slice has its own execution units (ALU and FPU). The slices are interconnected using routers that form a FAT-tree network. The top-level router is interconnected to four-leaf routers. Each leaf router is interconnected into four slices as shown in Figures 5.3 and 5.4. The FAT-tree network has the highest bandwidth at its root and the lowest bandwidth at its leaves. Scaling the number of

slices from 16 to 64 slices requires three levels of routers in a FAT-tree network with one top-level router, four mid-level routers, and sixteen leaf routers. Routing through the FAT-tree hierarchy is simple and efficient.

The M-Buffer feeds the programmed memory unit with memory instructions and scalar data. A memory instruction is broadcast by the routers to all the NS slices. The FAT-tree broadcast network feeds instructions and scalar data to all the slices.

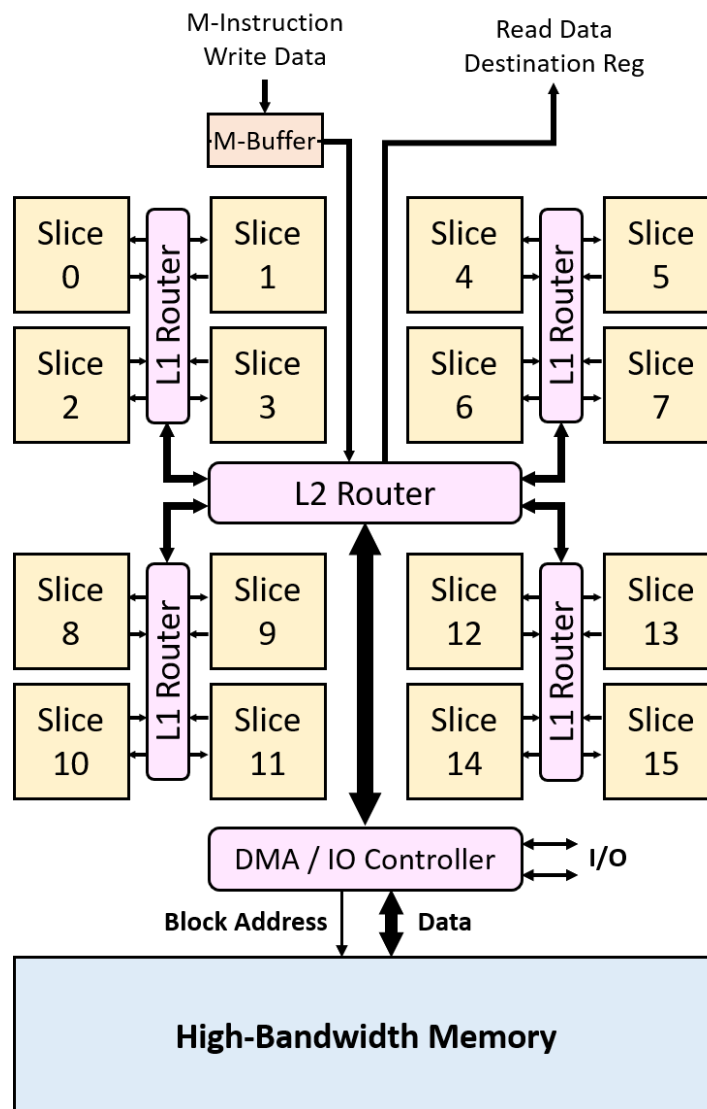


Figure 5. 3 Software Programmed Memory Diagram For Four Slices

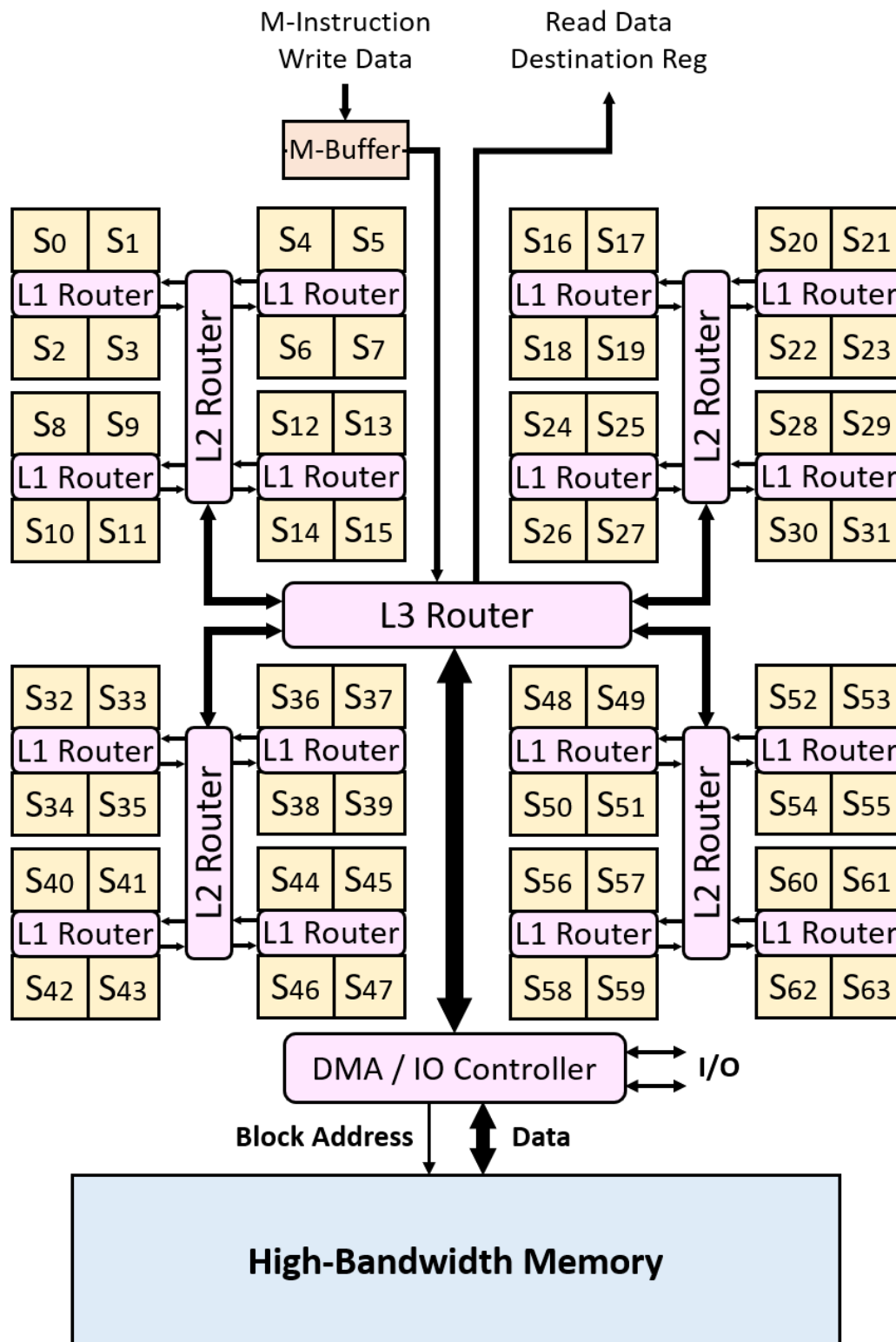


Figure 5. 4 Software Programmed Memory Diagram For 64 Slices

Load and store memory instructions are executed by the memory controller that transfers data efficiently between a high-bandwidth global memory and the memory slices. The high-bandwidth global memory can be implemented on the same substrate. It provides a very wide data bus to the top-level router, which distributes the data among its child routers. The distributed routers transfer the data between the memory controller and all the NS slices.

A memory computation instruction is executed in parallel in all the slices. A memory computation can be conditional. Some elements of a memory region are modified, while other elements are kept unchanged.

The FAT-tree reduction network transfers data across slices, starting at the leaf slices and moving up to the top-level router. The result of a reduction instruction is a scalar value that is transferred and stored in a scalar register.

A read memory instruction can read any slice of a memory region. A local address A is mapped and routed to slice $S = (A/8) \% NS$. An 8-byte octets is read from a memory region then transferred into a destination scalar register.

A write memory instruction can write any slice of a memory region, depending also on the address. For a write, the data is first transferred from a scalar register into the M-buffer, where the write instruction is queued. Then, the instruction is mapped and routed to its destination slice S , where the data is written in a specific target memory region.

A write memory instruction can also initialize some or all elements of a given memory region with the same scalar value. The FAT-tree broadcast network is used to initialize the elements in a memory region with the same scalar value across all the NS slices.

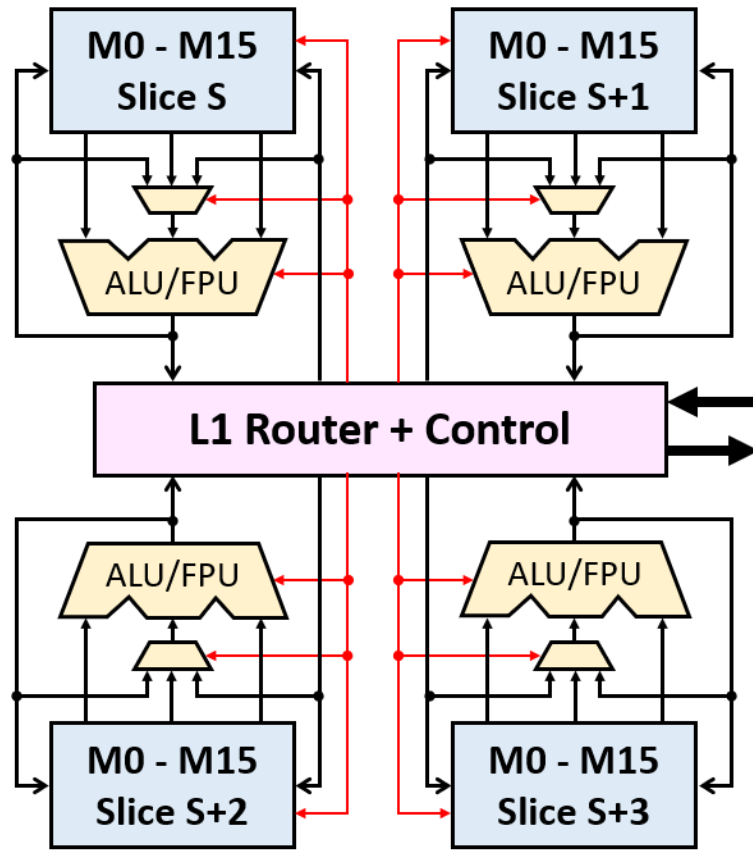


Figure 5. 5 Connecting four slices to a leaf router

The number of slices and the memory region length (MRL) are parameters that can be used to scale the Software Programmed Memory (SPM) according to the type of application. Increasing the MRL increases the number of slices that can hold more extensive data to perform many operations in parallel. Figure 5.3 can be scaled up to 64 slices to archive more parallelism as shown in Figure 5.4. The scalability of SPM will simplify the training processes in neural networks by training multiple inputs in a memory region that can be executed in parallel.

Therefore, the chapter described more details about our proposed Software Programmable Memory architecture used in this work for design data parallelism to support CNN and RNN with better performance.

CHAPTER 6

EXPERIMENT DESIGN AND SIMULATION

Machine learning is the process of training a machine in a particular skill to remember some data so that it can recognize or respond when there is an action in the input. The training process is simply adjusting the initial random weights of the signal to achieve better accuracy for a particular class input. Therefore, training a machine helps to have labels and fixed weight values that can be used to detect an input during testing. In this chapter, the experiments of the proposed training algorithms are explained, including the tools, parameters, and technology in general that were implemented during the training of the proposed CNN and RNN algorithms. The chapter has two main subsections. Section 6.1 explains the structure of the CNN and RNN algorithms with their parameters used during the experiment. Section 6.2 explains the proposed simulator used to simulate the proposed experiment.

6.1 Experiment Design

The goal of the experiment is to investigate the performance of the deep neural network algorithms for training multiple images in our proposed Software Programmable Memory (SPM) in M architecture. In this subsection, we describe the parameters and procedures used in the implementation of the CNN and RNN algorithms.

6.1.1 Convolutional Neural Networks (CNNs) Algorithm Implementation

The implementation of the CNN was developed according to Figure 6.1 to train multiple images in parallel. MNIST and CIFAR-10 are the input benchmarks that train their classes (i.e., ten images) at once in this study. The ten classes for MNIST and CIFAR-10 are shown in Figures 2.7

and 2.8, respectively. The experiment is designed to measure the performance of parallelism in training multiple images with supervised learning. Supervised learning is implemented by setting label accuracy values of each class, which are compared with the obtained computed output of SoftMax. The weight values of each neuron are recorded if the output values match the desired output, otherwise the process is repeated by adjusting the weights until they are closest or equal to the label. The number of repetitions to train the complete size of the input images from forward and backward propagation is called epochs. The CNN algorithm starts with the convolution operation between the input images and the kernel (i.e., the filter), as shown in Figure 6.1

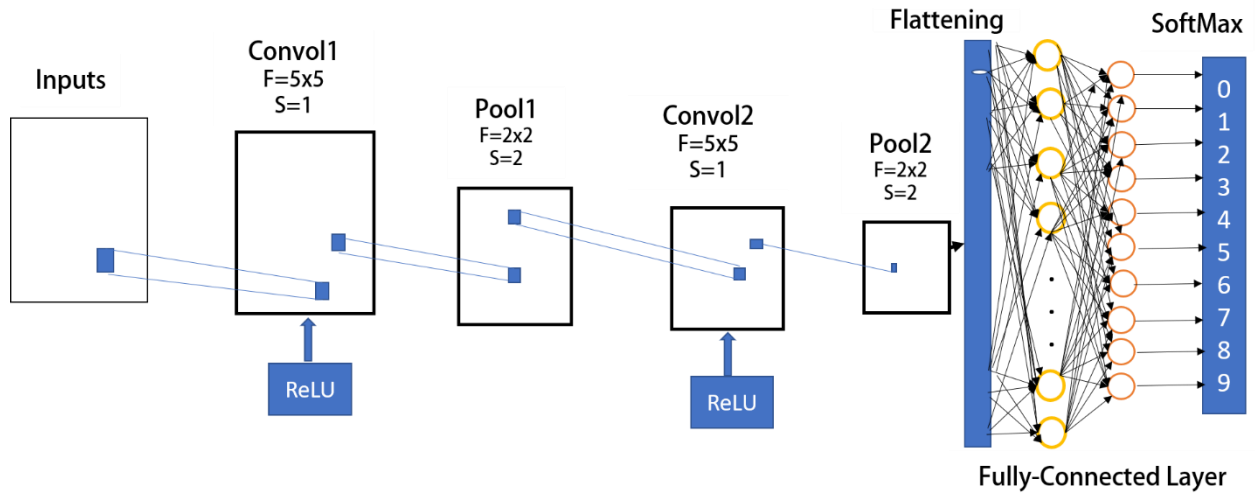


Figure 6. 1 The proposed Structure for CNN Algorithms

In this experiment, we investigate how the SPM can efficiently execute the CNN algorithms to improve the training performance. The process of the experiment includes information extraction and classification, where the extraction process that extracts information by reducing unwanted information is implemented by using a 5x5 kernel size in the experiment. The output of the extraction is used to classify an image in the fully connected layer. Table 6.1 describes the features and size in each CNN layer of the proposed experiment. In the first iteration, the value for initializing the label values for each class in the MNIST and CIFAR-10 benchmarks is set to 1 to

check the accuracy in training. The label values are selected according to the reference of the error rate of the benchmark [60]. The label values used in the backward propagation to determine the rate of error as shown in Equation 2.7, and Table 6.1 summarizes the computation of the size for each image in the benchmarks computed according to Equation 2.3.

The experiment was designed to implement ten images in one memory region since our SPM has a memory region length (MRL) of 8192, which is large enough to hold the MNIST size of ten images (i.e., $28 \times 28 \times 10 = 7480$) in one memory region, which is very beneficial for the proposed M architecture. The extraction of input values, Weight Initialization, and Kernel Selection are steps that are implemented before simulating the proposed instruction set.

Table 6. 1 The CNN's Output Layer Sizes after Operation

S. N	Benchmarks	MNIST	CIFAR-10
1.	Input Feature	28x28x1	32x32x3
2.	Convolution (Conv1) Padding = 0, Kernel = 5 x 5, stride = 1, and ReLU activation	24x24x1	28x28x1
3.	Pooling (Max-Pooling) Kernel Size = 2 x 2, and Stride = 2	12x12x1	14x14x1
4.	Convolution (Conv2) Padding = 0, Kernel = 5 x 5, stride = 1, and ReLU activation	8x8x1	10x10x1
5.	Pooling (Max-Pooling) Kernel Size = 2 x 2, and Stride = 2	4x4x1	5x5x1
6.	Flattening	16	25
7.	Hidden Layer (Denser 1)	10	10
8.	SoftMax	10	10

6.1.1.1 Extraction of Input Matrix Values

Matrix value extraction was implemented to obtain the matrix value of each image in the MNIST benchmarks for handwritten digits and CIFAR-10 RGB colour images. As a first step, the MNIST zip file named **train-images-idx3-ubyte** was downloaded from the official website [59] which has the following properties in Figure 6.2.

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803 (2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

Figure 6. 2 MNIST Benchmark Properties [59]

The second step is to collect the matrix value of the images by running the Python code for each image so that the matrix input of the image can be obtained. Figure 6.3 shows the Python code with the output of the matrix, which are row-wise pixels of image of digit number 4, but all other handwritten digit numbers were extracted individually in the same code. The pixel values obtained range from 0 as background (white) to 255 as foreground (black) in MNIST images.

CIFAR-10 is the colour photographs objects with the size of 32x32x3 pixels of 60000 images. The pixels of all colours are represented from 0 to 255. The depth 3 means that the three channels are red, green, and blue. Each colour has 1024 bytes, which sums the total size of the image to 3072 bytes. The dataset consists of ten classes known as [airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck]. In the CNN, the first convolution operation performs three convolutions for each image and sums them into one channel as shown in Table 6.1 and Figure 6.4.

[illegible]

Figure 6. 3 Python Codes for MNIST Matrix Extraction

```
In [16]: #This code is written by Ali Omar Ali (g201806200)
# The program extract the matrix of CIFAR-10 classes that I can use in my thesis
# Date Created: 15/02/2021
from PIL import Image

im = Image.open("images/0000.jpg")

In [2]: from matplotlib.image import imread
panda = imread("images/0001.jpg")
print(type(panda))

<class 'numpy.ndarray'>

In [6]: panda.size, panda.shape, panda.ndim

Out[6]: (3072, (32, 32, 3), 3)

In [26]: panda[:30]

Out[26]: array([[ 25,  17,  30],
 [ 23,  12,  20],
 [ 39,  23,  24],
 ...,
 [199, 152, 126],
 [201, 151, 126],
 [202, 152, 127]],
 [[ 30,  24,  38],
 [ 22,  12,  21],
 [ 30,  15,  18],
 ...,
 [211, 165, 141],
 [205, 159, 133],
```

Figure 6. 4 Python Codes for CIFAR-10 Matrix Extraction

6.1.1.2 Weight Initialization

The weight is the quantity value expressing the strength of the connection between two nodes, which can be either in the input to the hidden layer, or in the hidden layer to the hidden layer, or

the hidden to the output layer in the fully connected layer. There are many random methods to generate the weight using the normal distribution, but it depends on the activation function in the training [96]. According to [96][97] researches shows that initializing weight has points that need to be considered as follows:

- Initializing weight to 0 is a bad choice because all gradients in the layer will be identical (symmetry)
- Too small random weights shrink activation to zero along with the layers (vanishing gradient)
- Too large random weights multiply along with layers (exploding gradient)
- Weight should not be the same values
- Weight should have variances

The following are common weight initialization methods according to the activation function in the neural network experiment [98]:

- **He Initialization:** It's a random sample for ReLU activation.

Normal Distribution $N(0, \delta)$ where mean = 0 and δ = Standard deviation can be

$$\text{computed, } \delta = \sqrt{\frac{2}{fan_{in}}} \quad (6.1)$$

- **Glorot / Xavier Initialization:** (Sigmoid and tanh activation)

Normal Distribution $N(0, \delta)$ where mean = 0 and δ = Standard deviation can be

$$\text{computed, } \delta = \sqrt{\frac{2}{(fan_{in} + fan_{out})}} \quad (6.2)$$

Where: ***fan_in*** is the number of input units and ***fan_out*** is the number of output units.

Therefore, He initialization is used in the study because ReLU is the activation function selected in this experiment, as shown in Table 6.1. For this reason, the standard deviation for the full-connected layer as shown in Table 6.2 was calculated as follows:

- Flattening to Hidden layer (Dense 1):

The number of inputs for ten classes in MNIST (fan_{in}) = $16 \times 10 = 160$

$$\text{Therefore, } \delta(\text{MNIST}) = \sqrt{\frac{2}{fan_{in}}} = \sqrt{\frac{2}{160}} = 0.118$$

The number of inputs for ten classes in CIFAR-10 (fan_{in}) = $25 \times 10 = 250$

$$\text{Therefore, } \delta(\text{CIFAR-10}) = \sqrt{\frac{2}{fan_{in}}} = \sqrt{\frac{2}{250}} = 0.089$$

- Hidden layer to Output layer

The number of inputs for ten classes in MNIST (fan_{in}) = CIFAR-10 (fan_{in}) = 10

$$\text{Therefore, } \delta(\text{MNIST}) = \text{CIFAR-10 } (fan_{in}) = \sqrt{\frac{2}{fan_{in}}} = \sqrt{\frac{2}{10}} = 0.447$$

The standard deviation result is used to generate the initialization of the weights by the random generator tool [99]. This tool requires the number of weights, the mean, and the standard deviation. The number of weights is equal to the number of inputs times the number of outputs, as summarized in Table 6.2, and Figure 6.5 shows the example of the weights generated by the tool [99].

Table 6. 2 Weight Initialization Parameters

Layer	Properties	Benchmark	
		MNIST	CIFAR-10
Flattening to Hidden layer (Dense 1)	Mean & Standard Deviation	(0, 0.118)	(0, 0.089)
Hidden layer to Output layer		(0, 0.447)	(0, 0.447)
Flattening to Hidden layer (Dense 1)	Number of Weights	1600	2500
Hidden layer to Output layer		100	100

Gaussian Random Number Generator

You requested 1600 numbers from a Gaussian distribution with mean 0.0, standard deviation 0.118 and using 4 significant digits.

Here are your random numbers:

8.1130e-2	-1.1840e-1	5.4910e-2	5.4490e-2	2.4350e-2	1.6710e-1	1.7100e-1	-1.3450e-1
2.4740e-1	-1.7930e-1	1.2000e-3	-3.3350e-2	-1.5870e-2	-1.1740e-2	5.9590e-2	1.4980e-3
1.2440e-1	-9.2180e-2	-3.1250e-1	1.1020e-1	1.4480e-1	1.2270e-1	1.9360e-1	1.0620e-1
2.2150e-1	-7.4900e-2	-2.0040e-1	8.2650e-2	-1.2780e-1	1.7660e-2	2.1970e-1	-1.0910e-1
-4.9980e-2	-1.2450e-1	2.9190e-2	-9.2160e-2	1.4990e-2	-4.0980e-2	1.2710e-1	-7.8400e-2
1.5940e-1	1.8750e-1	8.6460e-2	-1.0850e-1	-8.1090e-2	8.6070e-2	-3.2940e-2	3.3500e-2
-2.9200e-1	4.8610e-3	1.0280e-1	-2.8470e-2	1.5600e-1	-8.1850e-2	-2.6220e-1	5.4390e-2
-5.0520e-2	1.1480e-1	1.1370e-1	-3.9160e-2	-6.4210e-2	2.4890e-2	-2.1730e-2	-6.1350e-2
-3.4220e-2	9.0500e-3	-1.4360e-1	3.8260e-2	1.8170e-1	3.4540e-2	2.7850e-3	-1.4180e-1
-1.5170e-1	-1.0180e-1	6.8520e-2	-1.1630e-1	-6.5650e-2	-1.5130e-1	-2.2720e-1	6.4960e-3
-1.4940e-1	3.4230e-2	-4.6820e-2	-1.0980e-2	5.4920e-2	-2.3120e-1	-2.8500e-2	5.6850e-2
5.0060e-2	3.5040e-2	-6.7730e-2	3.2150e-3	-1.9180e-2	1.4670e-3	-2.7650e-1	2.5870e-2
-3.8040e-2	2.4750e-1	-9.3190e-2	5.7740e-3	1.9320e-1	-7.4530e-2	-6.0920e-2	2.6200e-2
4.3880e-2	-3.7050e-2	4.1470e-2	-1.4300e-1	-1.5590e-1	-4.1970e-2	5.4420e-2	2.3470e-2
-1.0310e-1	-1.8700e-2	6.4060e-2	1.1430e-1	-3.9400e-2	-8.2830e-2	-5.7770e-2	-1.5590e-1
2.6980e-2	-1.1860e-4	2.6140e-1	6.6060e-2	1.3900e-1	3.7740e-2	3.1360e-1	-2.1230e-1
-1.1630e-1	5.6820e-2	1.5870e-2	9.1900e-2	2.8890e-1	-3.2820e-2	1.3690e-1	1.4320e-1
-9.4230e-2	7.3700e-2	-8.0410e-2	-5.4530e-2	-2.3580e-1	1.1490e-2	3.1040e-2	4.4700e-2
-5.6160e-2	-7.1780e-2	5.2150e-2	1.3360e-1	1.6060e-1	5.1970e-2	2.2250e-1	1.4570e-1
1.3230e-1	3.0450e-2	4.5810e-2	-1.7250e-1	-2.6160e-1	-3.0460e-2	-2.2360e-1	-1.2610e-1
7.6310e-2	1.4580e-1	-2.0980e-2	-1.1240e-1	-1.5180e-2	7.1100e-2	2.0570e-1	1.0330e-1
-1.3890e-1	5.9790e-3	-9.0990e-3	-2.0460e-1	-6.6490e-2	1.2940e-1	1.3520e-1	-3.5270e-2
-1.7220e-1	-1.5520e-1	-8.8380e-2	-1.5390e-2	7.3020e-2	3.6280e-2	-3.6270e-2	2.0650e-1
1.3520e-1	-2.4950e-1	1.3980e-1	-1.1240e-1	-7.7310e-2	6.4150e-2	1.5440e-1	-8.0650e-2
-8.1530e-3	-1.1590e-1	4.4650e-3	7.2240e-2	2.0850e-1	2.3350e-1	-1.7830e-1	3.7530e-3
9.6850e-2	-8.9790e-2	-2.0350e-2	1.1380e-1	-2.1530e-1	1.2430e-1	6.2920e-3	1.5770e-2

Figure 6. 5 Sample of the Matrix Weights values are Generated

6.1.1.3 Kernel Matrix

The kernel is the filter matrix (K) used to extract the feature map in the input. In this experiment, a 5x5 kernel is implemented as a space matrix according to the Toeplitz Matrix theorem, as shown in Figure 6.6 [64].

$$\begin{array}{c}
 \begin{array}{c} n \times n \\ \begin{pmatrix} X_1 & X_2 & X_3 \\ X_4 & X_5 & X_6 \\ X_7 & X_8 & X_9 \end{pmatrix} \end{array} \\
 \end{array}
 *
 \begin{array}{c}
 \begin{array}{c} m \times m \\ \begin{pmatrix} K_1 & K_2 \\ K_3 & K_4 \end{pmatrix} \end{array}
 \end{array}
 =
 \begin{array}{c}
 \begin{array}{c} 1 \times n^2 \\ \begin{pmatrix} X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 & X_8 & X_9 \end{pmatrix} \end{array}
 \times
 \begin{array}{c}
 n^2 \times (n-m+1)^2 \\
 \begin{pmatrix}
 K_1 & 0 & 0 & 0 \\
 K_2 & K_1 & 0 & 0 \\
 0 & K_2 & 0 & 0 \\
 K_3 & 0 & K_1 & 0 \\
 K_4 & K_3 & K_2 & K_1 \\
 0 & K_4 & 0 & K_2 \\
 0 & 0 & K_3 & 0 \\
 0 & 0 & K_4 & K_3 \\
 0 & 0 & 0 & K_4
 \end{pmatrix}
 \end{array}
 =
 \begin{array}{c}
 (n-m+1) \times (n-m+1) \\
 \begin{pmatrix} O_1 & O_2 & O_3 & O_4 \end{pmatrix}
 \end{array}
 \end{array}$$

Figure 6. 6 Convolution Simplification by using Toeplitz Matrix for the Kernel Matrix

3x3

0.91	0.32	0.07
0.73	0.26	0.81
0.53	0.68	0.14

5x5

0.27	0.64	0.44	0.84	0.29
0.28	0.06	0.89	0.99	0.33
0.64	0.67	0.08	0.38	0.03
0.04	0.31	0.16	0.57	0.08
0.87	0.85	0.97	0.71	0.96

Figure 6. 8 Python code to Generate Toeplitz Matrix for the Filter

The kernel is loaded in the form of a space matrix, which has a size of $(n - m + 1)^2$ by n^2 , where n is the size of the input image (i.e., $n = 28$), m is the size of the original filter, which is 5. The input size (X) of the MNIST image is $28 \times 28 \times 1$ bytes equal to 748 bytes for one image and 7480 bytes for ten images, which is less than the total size of each memory region of $8192 \times 8 \times 8 = 512\text{KiB}$. For CIFAR-10, each image is $32 \times 32 \times 3$ bytes equal to 3072 bytes = 3KiB, and for ten images it is 30KiB.

In summary, there is a trade-off in computing convolutional neural networks to normalize either all data in floating-point or integer. In [101], a CNN computation was explained in a floating-point number of 32 bits after normalizing the input pixels by dividing by 255 since the other parameters are of float data type. This calculation gives the most accurate result but is slower compared to the integer CNN after quantizing the weight and activation values. The computation is done at a better speed compared to the floating-point CNN, but the accuracy is lower compared to the floating-point CNN [101][102]. In this experiment, the integer (i.e., 1-byte integer) data type is used for each parameter instead of floating-point. Table 6.3 shows the data type of the parameters used in the experiment.

Table 6. 3 Data Type and Size of the Training Parameters

S. N	Parameter	Data Type	Element Size
1.	Input Feature, (x)		
	MNIST Dataset [61]	Integer	Byte
	CIFAR-10 Dataset [62]	Integer	Byte
2.	Weight, (w)	Integer	Byte
3.	Bias, (b)	Integer	Byte

6.1.2 Recurrent Neural Networks (RNNs) Implementation

The study implements recurrent neural networks to examine how the SPM can compute many inputs in parallel to improve the performance of the algorithm. Three sentences were formulated to compare the similarity between them according to equations (2.14) and (2.15) as the general computation of the RNN algorithm. The three sentences implemented in this research are "Hello, have a good day", "Hello, have a great day" and "Hello, this is RNN training", which were converted into an integer vector using the 1-hot method and word2vec, as shown in Table 6.4 and Figure 6.10, respectively.

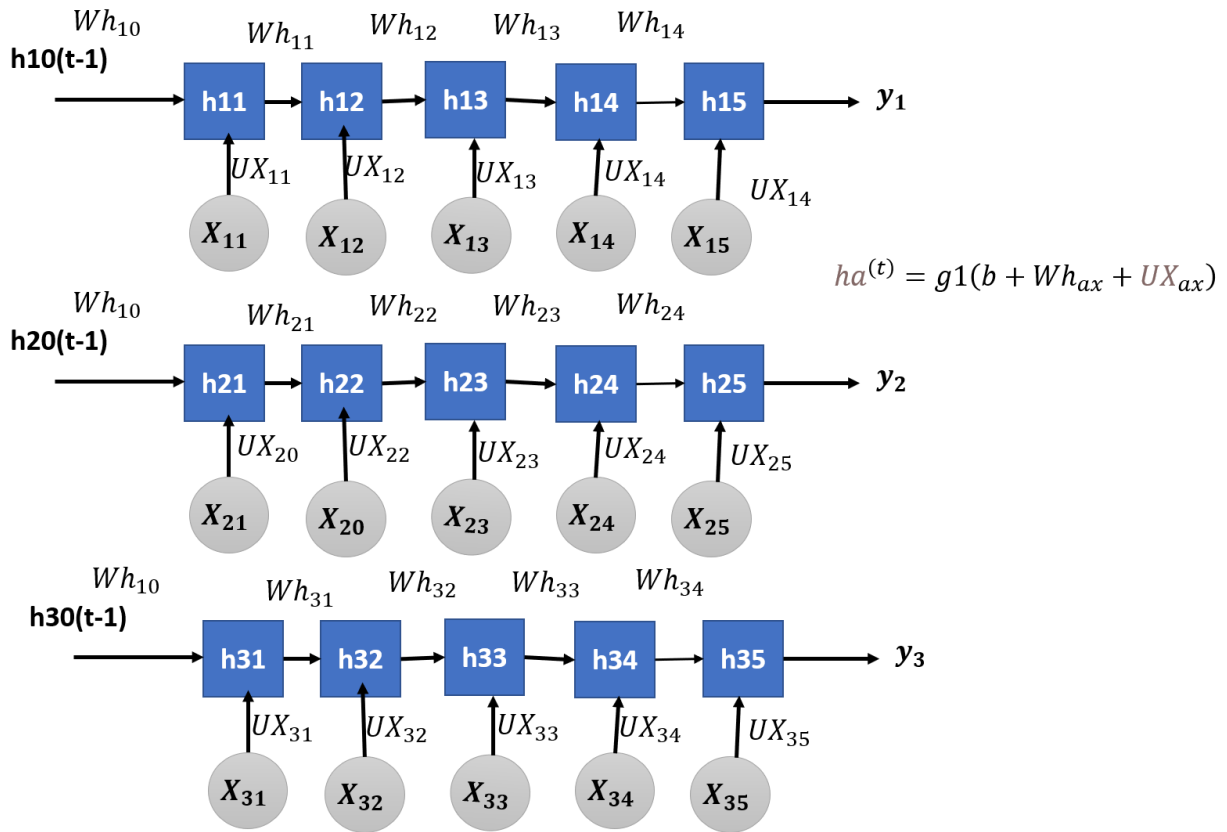


Figure 6. 9 Implementation Design of the RNNs Diagram for the Three Sentences

Table 6. 4 The Selected Sentences in 1-Hot Representation

Sentence	1-Hot Representation
<i>“Hello, Have a good day”</i>	[90, 56, 84, 14, 69]
<i>“Hello, Have a great day”</i>	[90, 56, 84, 31, 69]
<i>“Hello, This is RNN training”</i>	[90, 37, 3, 97, 21]

The vector of each sentence has been generated through Python script as shown in Figure 6.9 as input of the computation by multiple with weight matrix one by one respectively. Figure 6.10 shows the Python script used to convert sentences into vectors through word2vec. The computation was implemented in parallel by loading all three input sentences into one memory region (e.g., M0) in the first step. In the second step, the weighting values between inputs and hidden layers (i.e., U and W) are loaded into the next memory region. Then, the multiplication between the input X and the weighting U is calculated in parallel for all the sentences.

```
In [15]: #This code is written by Ali Omar Ali (g201806200)
# The program extract the matrix of word2vec classes that I can use in my thesis
# Date Created: 15/02/2021

from numpy import array
from keras.preprocessing.text import one_hot
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding

# define document
#define the sentences below:
tokenized_sentences = ['Hello, Have a good day', 'Hello, Have a great day',
                      'Hello, This is RNN training']

#define classes label
sent_labels = array([1,1,0])

# integer encoding of the documents
my_vocab_size = 100
encoded_sent = [one_hot(i, my_vocab_size) for i in tokenized_sentences]
print(encoded_sent)

[[90, 56, 84, 14, 69], [90, 56, 84, 31, 69], [90, 37, 3, 97, 21]]
```

Figure 6. 10 Python script for Generate 1-Hot Word Embedding

```

1 #This code is written by Ali Omar Ali (g201806200)
2 # The program extract the matrix of word2vec classes that I can use in my thesis
3 # Date Created: 15/02/2021
4 #define the sentences below:
5 tokenized_sentences=[['Hellow', 'Have', 'a', 'good', 'day'], ['Hellow', 'Have', 'a', 'great', 'day'],
6                       ['Hellow', 'This', 'is', 'RNN', 'training', 'by', 'Ali']]
7 # word Embedding (word2vec)
8 import logging
9 from gensim.models import Word2Vec
10 mymodel = Word2Vec(tokenized_sentences, min_count=1)
11 #print the summare of the model
12 print(mymodel)
13 #word to vector
14 words = list(mymodel.wv.vocab)
15 print(words)
16 #access word vector for each word
17 print(mymodel['Hellow'])
18
Word2Vec(vocab=12, size=100, alpha=0.025)
['Hellow', 'Have', 'a', 'good', 'day', 'great', 'This', 'is', 'RNN', 'training', 'by', 'Ali']
[-4.6194326e-03  2.7277637e-03  3.2264814e-03  4.6178331e-03
  4.5049200e-03 -1.9074999e-03  3.9302697e-03  5.5223581e-04
  3.7606205e-03 -2.8521122e-04  6.4469053e-04 -2.8601708e-03
  3.3921511e-03 -4.0338347e-03  4.2567365e-03 -9.0433029e-04
 -2.4827451e-03 -1.5776996e-03 -4.2410740e-03 -4.6862075e-03
  4.6862075e-03 -4.2410740e-03 -1.5776996e-03 -2.4827451e-03
  3.3921511e-03  4.2567365e-03  4.0338347e-03  3.7606205e-03
  5.5223581e-04  3.9302697e-03  1.9074999e-03  4.6178331e-03
  3.2264814e-03  2.7277637e-03  4.6194326e-03]

```

Figure 6. 11 Python script for Generate Vector in Word2Vec Method

6.2 Simulation of the Experiment

The experiment is performed in simulation mode using M-Simulator, which we developed for simulating M- Architecture Instructions Set (ISA). M-Simulator is a high-level abstract simulator that simulates the functionality of our instructions and their performance matrices such as number of instructions (scalar or memory), number of operations, total cycle, and number of operations per cycle. Figure 6.12 shows the user interface of the M simulator. The simulator is flexible and dynamic to support all requirements. Only the initial configuration parameters such as global memory size, local memory size, registers, number of execution units need to be set.

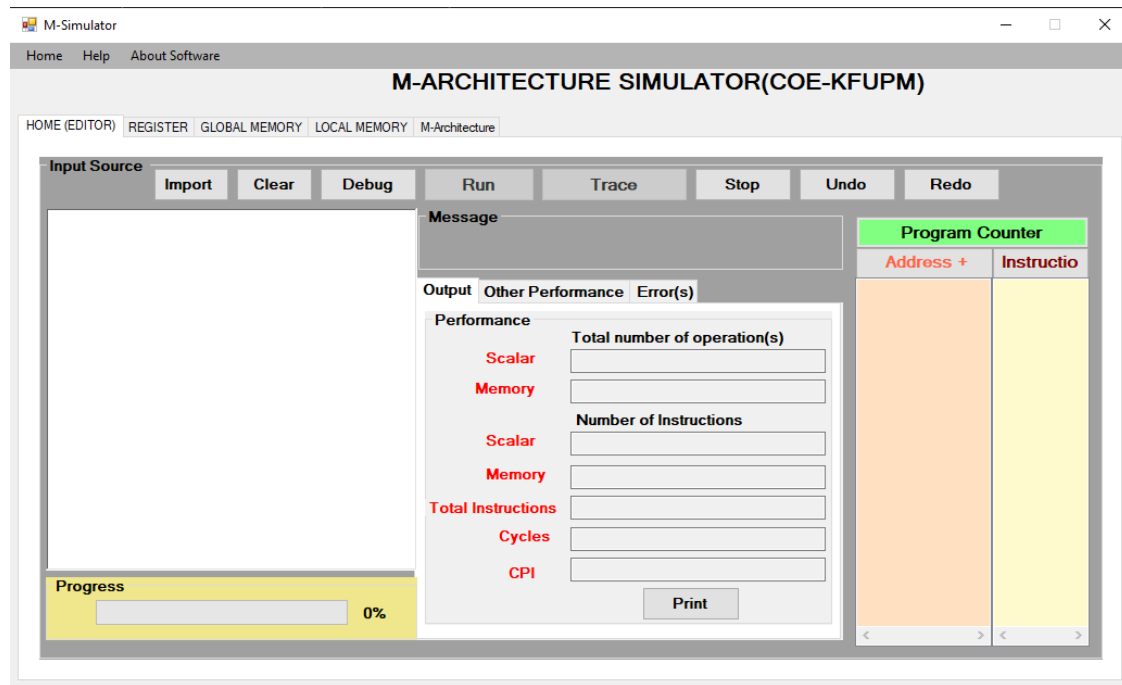


Figure 6. 12 M-Simulator User Interface (UI)

The performance metric of the benchmarks evaluated using the number of operations, number of instructions, and latency time. M-Simulator has been encoded the instructions cycles for our instruction as shown in Table 6.5.

Table 6. 5 Performance Equations Mode in M-Simulator

Instruction Type	Instruction Cycles
Integers ALU	1
Integer Multiply	3
Integer Divide	12
Floating-Point	5
Floating Point Divide	12
Local memory read write	1
Memory latency	10 – 20
Memory Transfer Cycles	Length/memwidth
Memory Load	Memory Latency +transfer cycles
Memory Operation	Length/NFU

The work started by configuring the parameters such as memory region length (MRL), size of elements, number of slices (NS) which help to achieve the performance metric of the proposed benchmark. Table 6.6 shows our configurable parameter for this work.

Table 6. 6 Configurable Parameters for our Experiment

Parameter	Default Value
Memory Region Length (MRL)	$2^{15} = 32768$ Bytes
Number of Slices (NS)	1, 4, 8, 16, 32, 64
Execution Unit per Slices (NEU)	ALU8X8, ALU16X4, ALU32x2, ALU64, MUL64, FPU32x2, FPU64
Memory Width (memwidth)	NS bytes/cycle
Local Memory Region (LMR)	16

The model equations and the parameters help the simulation to achieve the performance of the benchmark. Figure 6.13 shows the sample of the configurable parameter that has been set during the experiment.

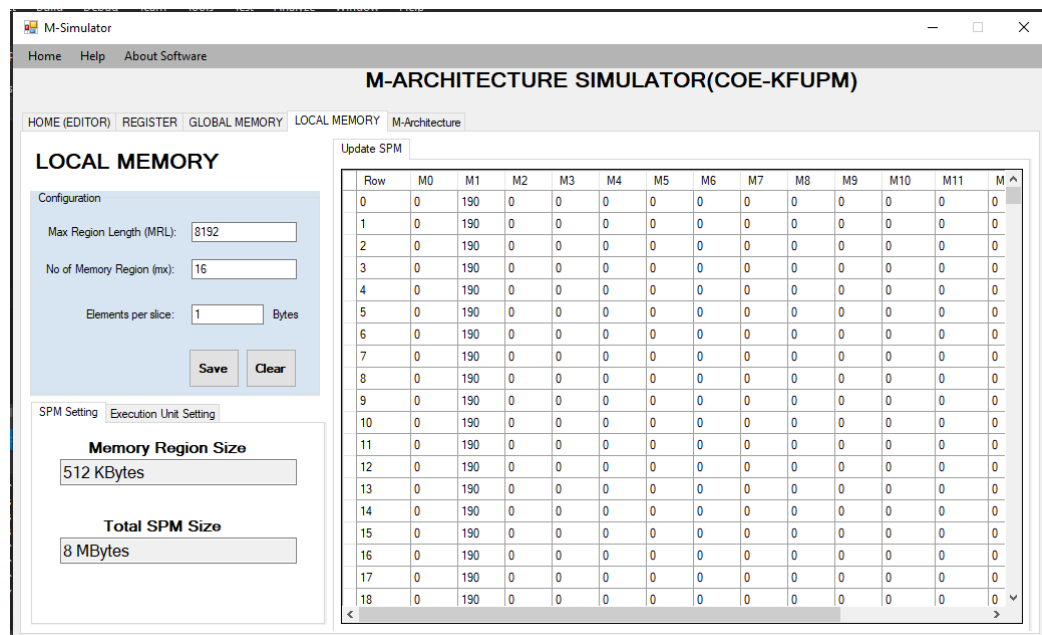


Figure 6. 13 Sample of the Configurable Parameters in M-Simulator

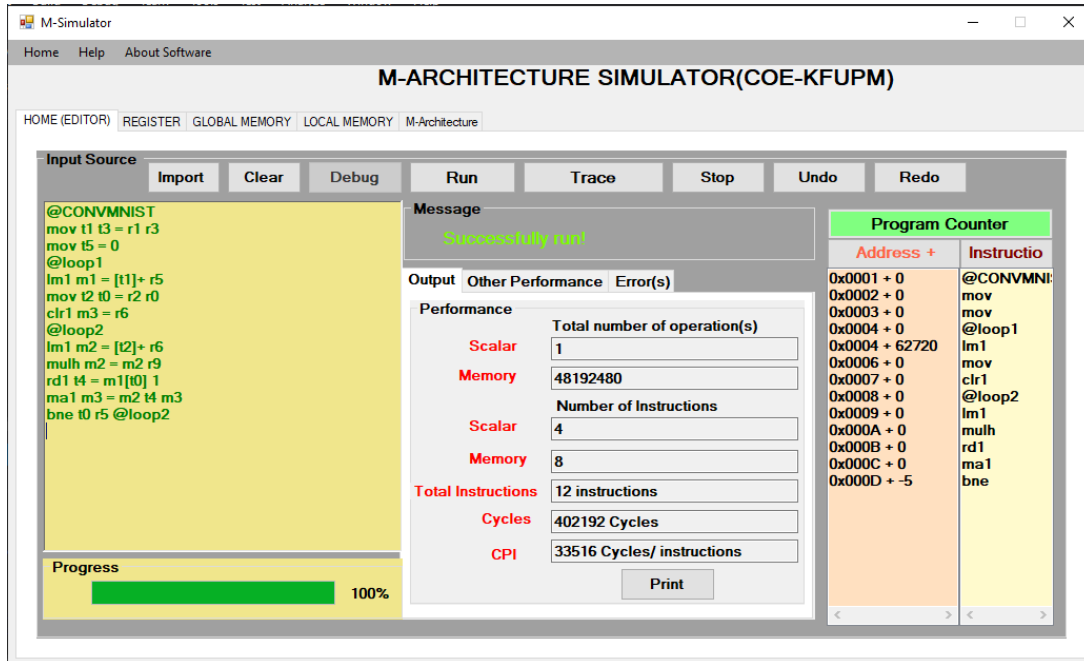


Figure 6. 14 Sample of the Running Experiment (Convolution)

Figure 6.14 demonstrates the implementation convolution operation of the work during the experiment. We have written the code into M-Simulator editors, then we debug to check the correctness of our instruction, then we run to record the performance.

6.3 Verification of the M-Simulator

M-simulator is developed to check the functionality and performance of the M-Instruction Set (ISA). The instruction is written in high-level language through C# programming in visual studio to verified each instruction's functionality and performance. In this work, the proposed instruction set has been tested by comparing it with python programming. Figure 6.15 shows the sample of matrix used during verification of our instructions by running it in our M-simulator and Python. We choose a small size matrix to get easily tracking the operation result. The size of the matrix does not affect the verification because it's just a parameter that can be scalable in any size.

$$\mathbf{X} = \begin{matrix} & \mathbf{4 \times 4} \\ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} & \mathbf{K} = \begin{matrix} & \mathbf{3 \times 3} \\ \begin{pmatrix} 0.91 & 0.32 & 0.07 \\ 0.73 & 0.26 & 0.81 \\ 0.53 & 0.68 & 0.14 \end{pmatrix} & = \frac{1}{256} \begin{matrix} & \mathbf{3 \times 3} \\ \begin{pmatrix} 233 & 82 & 18 \\ 187 & 67 & 207 \\ 136 & 174 & 36 \end{pmatrix} \end{matrix} \end{matrix}$$

Figure 6. 15 Verification Input Matrix (X) and Filter Matrix (K)

```

1 # example of vertical line detection with a convolutional layer
2 # Ali Omar Ali (g201806200)
3 from numpy import asarray
4 from keras.models import Sequential
5 from keras.layers import Conv2D
6 # define input data
7 data = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 1, 2, 3], [4, 5, 6, 7]]
8 data = asarray(data)
9 data = data.reshape(1, 4, 4, 1)
10 # create model
11 model = Sequential()
12 model.add(Conv2D(1, (3,3), activation='relu', input_shape=(4, 4, 1)))
13 # summarise model
14 model.summary()
15 # define a vertical line detector
16 detector = [[[[[0.91]], [[0.32]], [[0.07]]], [[[[0.73]], [[0.26]], [[0.81]]], [[[[0.53]], [[0.68]], [[0.14]]]]]]
17 weights = [asarray(detector), asarray([0.0])]
18 # store the weights in the model
19 model.set_weights(weights)
20 # apply filter to input data
21 yhat = model.predict(data)
22 # enumerate rows
23 for r in range(yhat.shape[1]):
24     # print each column in the row
25     print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

Model: "sequential_5"
Layer (type)                Output Shape                Param #
-----
conv2d_5 (Conv2D)           (None, 2, 2, 1)            10

Total params: 10
Trainable params: 10
Non-trainable params: 0

WARNING:tensorflow:6 out of the last 6 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7f1a810e20e0> triggered tf.func
[18.37, 18.050001]
[21.77, 18.65]

```

Figure 6. 16 Sample of the Convolution Operation in Python

Figure 6.16 shows the simulation of convolution operation in python. The same matrix is simulated with our proposed instruction set. In M-simulator, the simulation of convolution operation is implemented in terms of vector-matrix multiplication by treating the input matrix as a one-row vector and filter matrix into the Toeplitz matrix as shown in Figure 6.17.

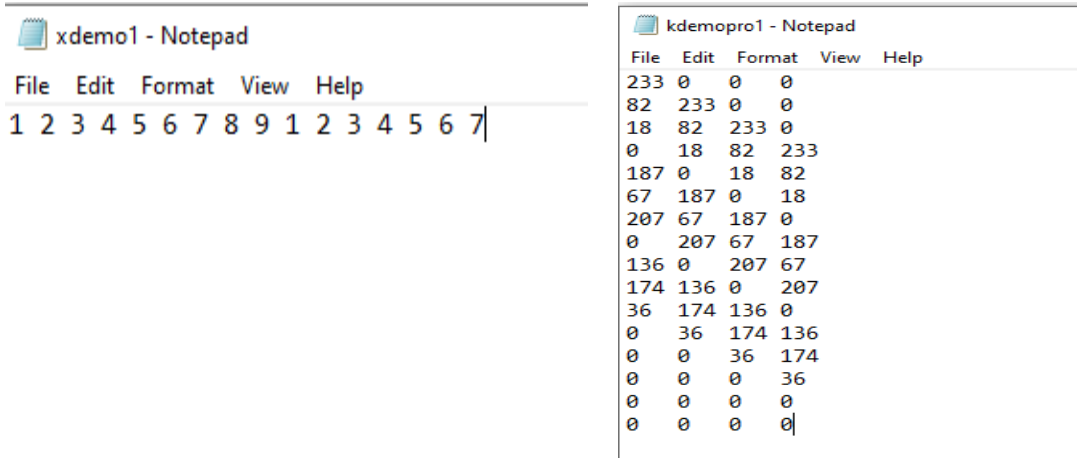


Figure 6. 17 Input Vector (X) and Kernel Toeplitz Matrix (K) for M-Simulator

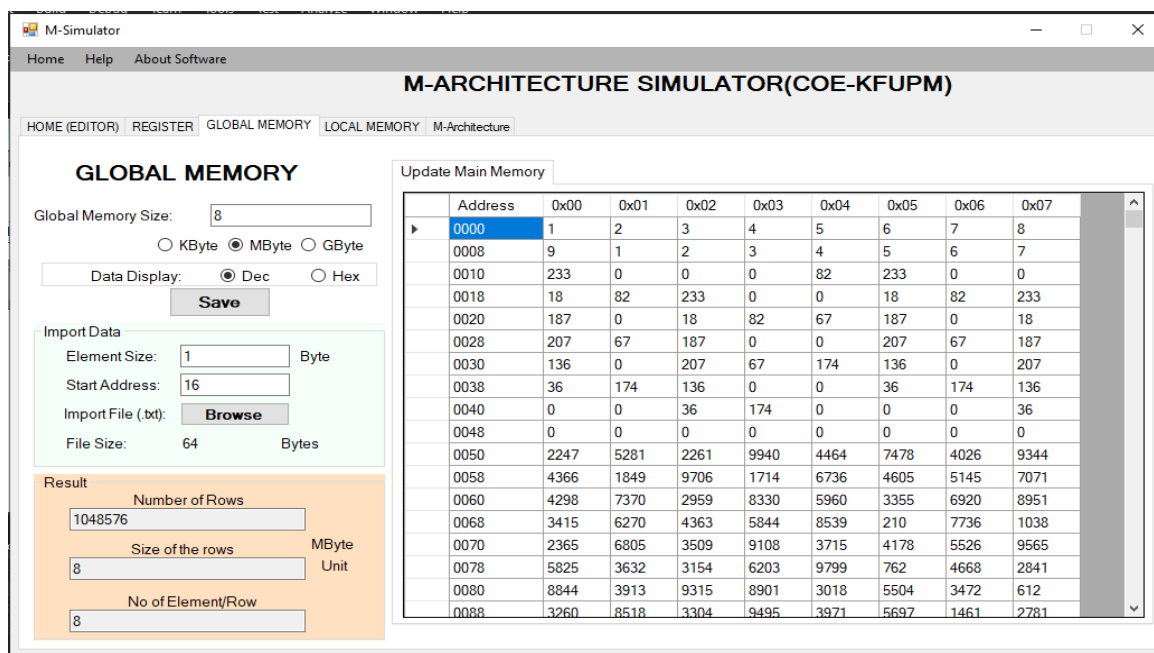


Figure 6. 18 Loaded of the Input X and Kernel Matrix Byte Elements in Global Memory

Figure 6.18 shows the imported byte-elements (i.e., input vector, X and kernel matrix, K) in the global memory. Figure 6.19 shows the running of our proposed instruction set within in M-simulator and its performance. Figure 6.20 shows the computation result after completed the simulation for convolution operation.

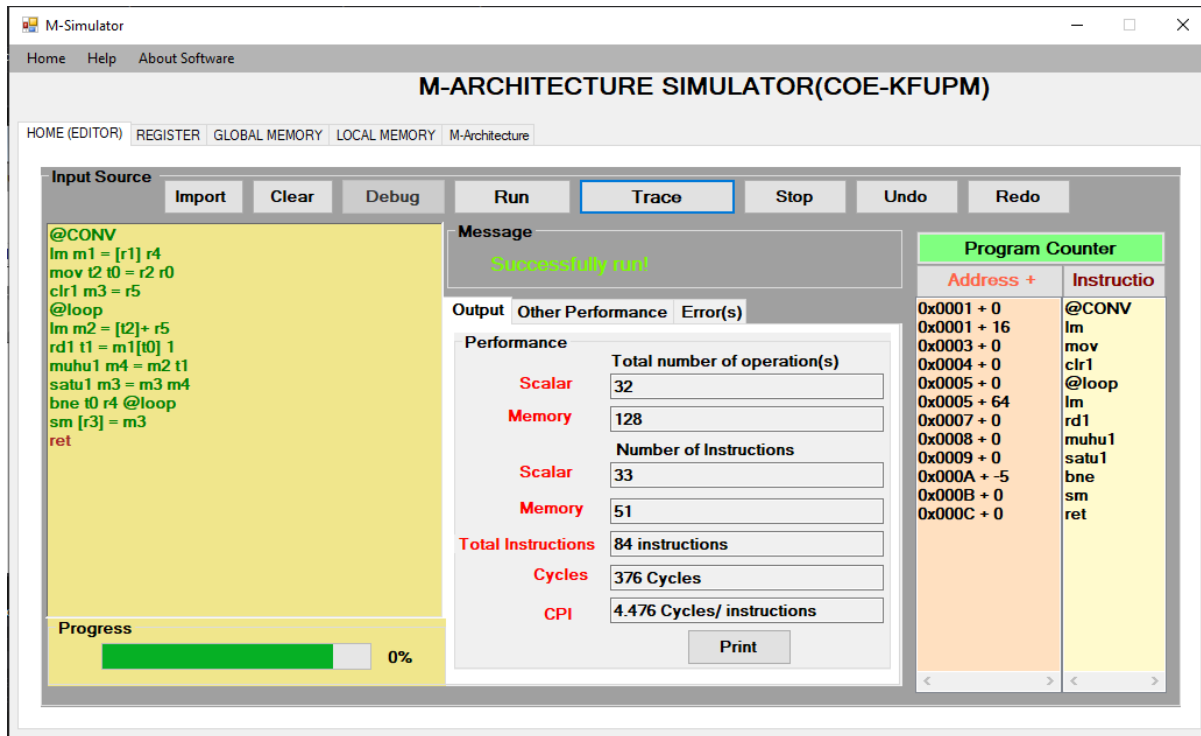


Figure 6. 19 Sample of the Convolution Operation in M-Simulator

The performance of the simulation has been verified by mathematical analysis. The operation involved 16 iterations in the loop to multiplying the scalar input byte by each row in 64-byte elements of the filter. The mathematical verification as follows:

Number of Instructions:

Scalar Instructions = Before Loop + In the Loop + After Loop

$$= \text{mov} + 16 \times (\text{bne}, \text{rd1}) + 0 = 1 + 2 \times 16 = 33 \text{ Instructions}$$

Memory Instructions = Before Loop + In the Loop + After Loop

$$= (\text{lm}, \text{clr1}) + 16 \times (\text{lm}, \text{muhu1}, \text{satu1}) + \text{sm} = 2 + 16 \times 3 + 1 = 51$$

Therefore, the total Instruction = 51 + 33 = 84 Instructions.

Number of Cycles = Total Cycles before Loop + Cycles in the Loop + Cycles after Loop

$$= C (lm + mov + clr1) + 16 \times [lm + rd1 + muhu1 + satu1 + bne] + sm$$

$$= (11+1+1) + 16 \times (11+1+3+1+6) + 11$$

$$= 376 \text{ cycles}$$

$$\text{Cycles Per Instruction (CPI)} = \frac{\text{Cycles}}{\text{Instructions}} = \frac{376}{84} = 4.476 \text{ cycles per instruction}$$

Number of Operations:

$$\text{Scalar Operations} = 16 \times (bne + rd1) = 16 \times 2 = 32 \text{ operations}$$

$$\text{Memory Operations} = 16 [(4 \text{ bytes in } muhu1) + (4 \text{ bytes in } satu1)] = 128 \text{ operations}$$

Therefore, the performance obtained in M-Simulator gives the same result by mathematical verification.

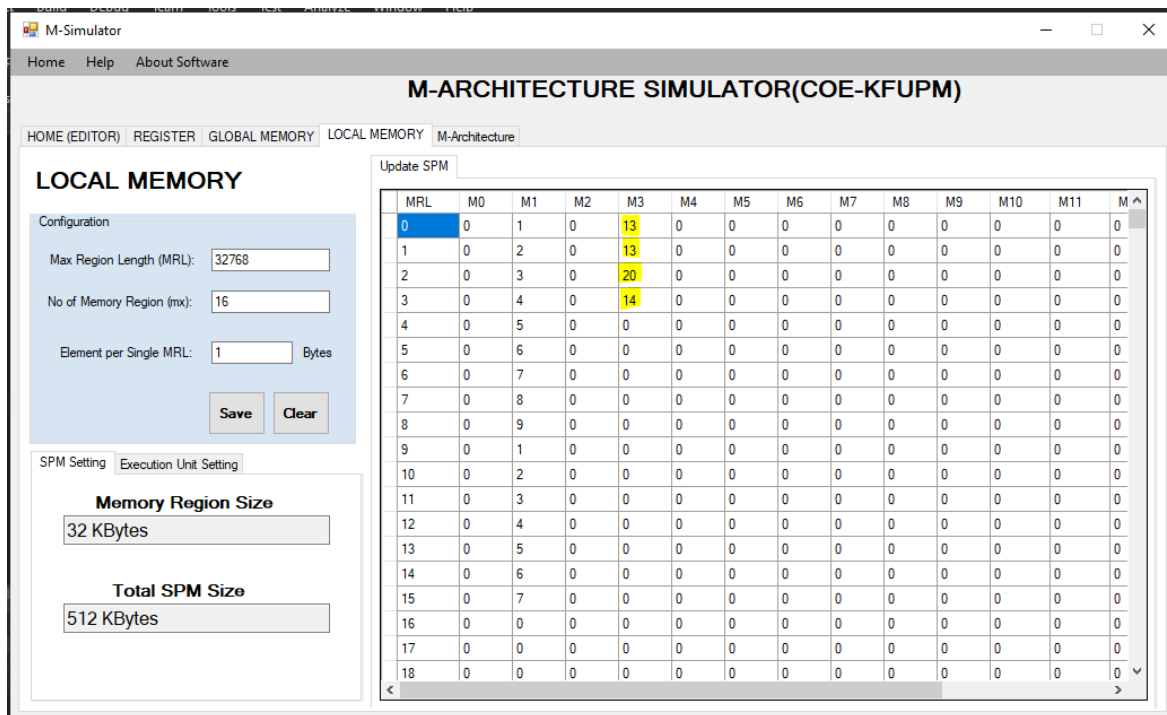


Figure 6. 20 Result of the Convolution Operation in M-Simulator

The output result in M-simulator gives less accuracy compared with python due to the conversion of kernel elements into a 1-byte integer from floating-point. Not only that, our proposed **muhu1** and **satu1** instructions affect the computation accuracy because muhu1 instruction multiplies the unsigned byte elements, stores the **high-byte** of the product, and the low-byte of the product is discarded. The satu1 instruction adds the **unsigned bytes** and saturates the sum at **255**. Consider the following computation shows the effect of muhu1 instruction set.

Operation (X*K)	Result	Muhu1 Instruction
(1x233)/256	= 0.91	= 0
(2x82)/256	= 0.64	= 0
(3x18)/256	= 0.21	= 0
(5x187)/256	= 3.65	= 3
(6x67)/256	= 1.57	= 1
(7x207)/256	= 5.66	= 5
(9x136)/256	= 4.78	= 4
(1x174)/256	= 0.68	= 0
(2x36)/256	= 0.28	= 0
Total	= 18.38	= 13

The result in the computation verifies the relationship between the output in the python and the M-Simulator as shown in Figure 6.16 and Figure 6.20 respectively. However, the other instructions provided the same result when did not involve muhu1 or satu1. Figure 6.21 and Figure 6.22 show the same result of the maximum pooling operation in Python and M-simulator.

G201806200.ipynb

File Edit View Insert Runtime Tools Help Last edited on May 2

Code + Text

```

1 # example of max pooling
2 # Ali Omar Ali (g201806200)
3 from numpy import ndarray
4 from keras.models import Sequential
5 from keras.layers import Conv2D
6 from keras.layers import MaxPooling2D
7 # define input data
8 data = [[0, 0, 0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0],
9         [0, 0, 0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0]]
10 data = ndarray(data)
11 data = data.reshape(1, 8, 8, 1)
12 # create model
13 model = Sequential()
14 model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
15 model.add(MaxPooling2D())
16 # summarise model
17 model.summary()
18 # define a vertical line detector
19 detector = [[[[[0]], [[1]], [[0]]], [[[[0]], [[1]], [[0]]], [[[[0]], [[1]], [[0]]]]], [[[[0]], [[1]], [[0]]]]]]
20 weights = [ndarray(detector), ndarray([0.0])]
21 # store the weights in the model
22 model.set_weights(weights)
23 # apply filter to input data
24 yhat = model.predict(data)
25 # enumerate rows
26 for r in range(yhat.shape[1]):
27     # print each column in the row
28     print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10
max_pooling2d (MaxPooling2D)	(None, 3, 3, 1)	0

Total params: 10
Trainable params: 10
Non-trainable params: 0

```

[0.0, 0.0, 0.0]
[0.0, 0.0, 0.0]
[0.0, 0.0, 0.0]

```

Figure 6. 21 Sample of Maximum Pooling Operation in Python

M-Simulator

Home Help About Software

M-ARCHITECTURE SIMULATOR(COE-KFUPM)

HOME (EDITOR) REGISTER GLOBAL MEMORY LOCAL MEMORY M-Architecture

Input Source Import Clear Debug Run Trace Stop Undo Redo

```

@MaxPool
clr1 m2 = r4
@loop1
lm m0 r2 = [r1]+ r3
cmax m1 = m0 2
bne r2 r3 @loop1
rmax m2 = m1 2
sm [r6]+ = m2
ret

```

Progress 0%

Message: Successfully run!

Output Other Performance Error(s)

Performance

	Total number of operation(s)
Scalar	1
Memory	45
Scalar	1
Memory	5
Total Instructions	6 instructions
Cycles	31 Cycles
CPI	5.167 Cycles/ instructions

Print

Program Counter

Address +	Instructio
0x0001 + 0	@MaxPool
0x0002 + 0	clr1
0x0003 + 0	@loop1
0x0003 + 36	lm
0x0005 + 0	cmax
0x0006 + -3	bne
0x0007 + 0	rmax
0x0008 + 0	sm
0x0009 + 0	ret

Figure 6. 22 Maximum Pooling Operation in M-Simulator

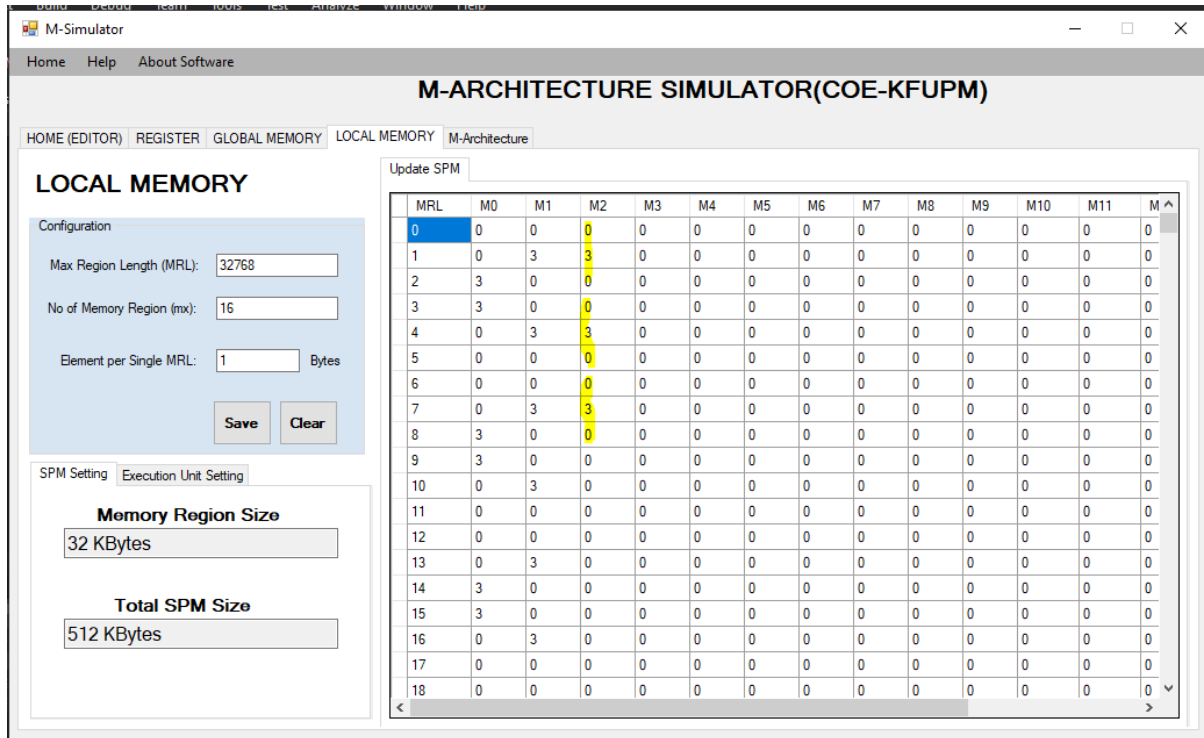


Figure 6. 23 The Output Result of Maximum Pooling Operation in M-Simulator

Therefore, M-simulator provides the approximation output based on the operation involves the satu1, sat1, muh1, and muhu1 instructions. The performance of the work is verified and validated as correctly estimated based on the mathematical analysis.

In summary, this chapter explains experiments of the proposed training algorithms including the tools, parameters, and technology in general that were implemented during the training of the proposed CNN and RNN algorithms. The chapter also explains the proposed simulator used to simulate the proposed experiment to verify the functionality and performance computation such as number of operations, number of cycles, and number of operations per cycle.

CHAPTER 7

BENCHMARK RESULTS

This chapter described the implementation of the proposed benchmarks by evaluating and analysing their performance within Software Programmable Memory (SPM). All the proposed deep neural network benchmarks were tested to measure a wide range of data-level parallelism within SPM. The chapter is divided into subsections to discuss each benchmark separately according to its algorithm group. The proposed instructions with the experimental results of the proposed benchmarks are discussed in this chapter.

7.1 Convolution Neural Networks

In this subsection, we describe the training of CNN operations from convolution, max-pooling, fully-connected-layer, SoftMax, and backward propagation according to Figure 6.1. The description is based on execution, verification, and evaluation in the M-architecture. MNIST and CIFAR-10 have been implemented as benchmarks for training the CNN. The description of each CNN operation follows:

7.1.1 Convolution Operation

The convolution operation is the sliding dot product between the filter and the input to extract the feature map. The sliding process reduces the power of the operation in hardware. In this situation, we implemented matrix multiplication between them instead of sliding the filter within the input. Figure 7.1 shows the conversion of the kernel matrix to a space matrix and the overall matrix multiplication convolution that we adapted in this work. This operation is implemented twice in

the experiment (i.e., CONV1 and CONV2) because the first CONV1 learns patterns and edges, and the second CONV2 tries to understand the shape/colour and other features. We started to implement the benchmark one by one with the selected kernel filter, as shown in Figure 6.7. In these steps, we ran two experiments separately between the MNIST and CIFAR-10 datasets. The input X of the benchmarks loaded their classes (i.e., ten images) into one memory region.

$$\begin{array}{c}
 \begin{array}{c} n \times n \\ \begin{pmatrix} X_1 & X_2 & X_3 \\ X_4 & X_5 & X_6 \\ X_7 & X_8 & X_9 \end{pmatrix} \end{array} \\
 \end{array}
 *
 \begin{array}{c}
 \begin{array}{c} m \times m \\ \begin{pmatrix} K_1 & K_2 \\ K_3 & K_4 \end{pmatrix} \end{array} \\
 \end{array}
 \\
 \\
 \begin{array}{c}
 \begin{array}{c} 1 \times n^2 \\ \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 & X_8 & X_9 \\ \hline \end{array} \end{array} \\
 \times
 \begin{array}{c}
 \begin{array}{c} n^2 \times (n-m+1)^2 \\ \begin{pmatrix} K_1 & 0 & 0 & 0 \\ K_2 & K_1 & 0 & 0 \\ 0 & K_2 & 0 & 0 \\ K_3 & 0 & K_1 & 0 \\ K_4 & K_3 & K_2 & K_1 \\ 0 & K_4 & 0 & K_2 \\ 0 & 0 & K_3 & 0 \\ 0 & 0 & K_4 & K_3 \\ 0 & 0 & 0 & K_4 \end{pmatrix} \end{array} \\
 \end{array}
 =
 \begin{array}{c}
 \begin{array}{c} (n-m+1) \times (n-m+1) \\ \begin{array}{|c|c|c|c|} \hline O_1 & O_2 & O_3 & O_4 \\ \hline \end{array} \end{array}
 \end{array}
 \end{array}$$

Figure 7. 1 Convolution Implementation in SPM

The inputs of the image pixels are **unsigned bytes**. The space matrix carries **weights** that can be represented by a matrix of **unsigned bytes** from **0** to **255** and map to **0** to **0.996** when divided by **256**. The neurons at the outputs of a convolution compute the weighted average of their inputs, using matrix-matrix multiplication that saturates results at **255**. The saturated weighted averages pass through a **ReLU** activation function that reduces all values smaller than **128** (or **0.5**) down to **0**. The output of the activation function is either **0** or in the range **128** to **255**. Figure 7.2 shows more clearly how we implemented convolution as matrix multiplication. The number of rows in the matrix X represents a number of images, where each row contains 28x28 bytes for MNIST and 32x32 for the CIFAR-10 benchmark. In the MNIST benchmark, the size of matrix X is equal to 10x784 bytes = 7840 bytes (~ 8 KiB), and 784x576 bytes (441 KiB) for kernel matrix K whereby

the size of the memory region is 8192x8x8 bytes (512 KiB). Therefore, the matrix X rows fit in a memory region ($n \leq \text{MRL}$) and rows of K also fit ($c \leq \text{MRL}$).

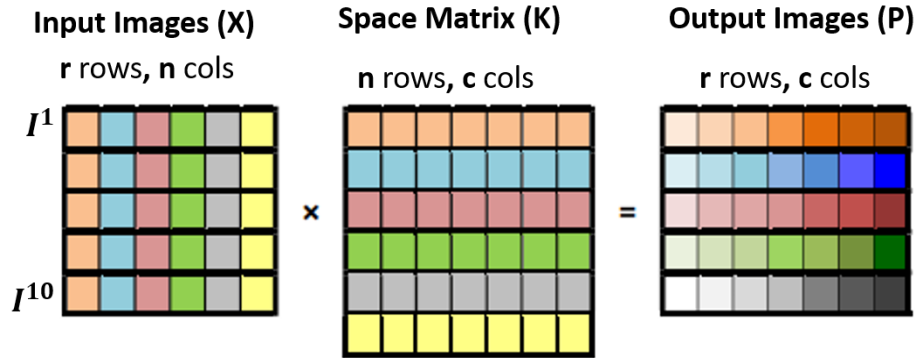


Figure 7. 2 Matrix-Matrix Multiplication for Convolution Implementation

The following **CONVMINST** function multiplies an input vector **X** of length **r** images and **n** bytes by a matrix **K** with **n** rows and **c** columns to produce an output product matrix **P** of length **r** images and **c** bytes that saturate results at **255**. The product matrix passes through an activation function that reduces all values below **128** down to **0**.

```
# CONVMINST: Convolution Matrix Matrix Multiply (1-byte elements)
# r1: Address of matrix X in global memory
# r2: Address of matrix K in global memory
# r3: Address of product matrix P in global memory
# r4: Number r of rows in matrix X = Number of rows in matrix P
# r5: Number n of columns in matrix X = Number of rows in matrix K
# r6: Number c of columns in Matrix K= Number of columns in matrix P
# This version assumes that  $n \leq \text{MRL}$  and  $c \leq \text{MRL}$ 
```

```
@CONVMINST
```

```
mov t1, t3 = r1, r3
```

```
mov t5 = 0
```

```
@loop1
```

```
lm m1 = [t1]+ r5
```

```
mov t2, t0 = r2, r0
```

```
clr1 m3 = r6
```

```
@loop2
```

```
# CONVMINST Function
```

```
# t1 = address of X, t3 = address of P
```

```
# t5 = row index i in X and P (Image)
```

```
# Loop over the I images of matrix X
```

```
# Load m1 = row i of X; t1+=c
```

```
# t2 = address of K, t0 = index j=0
```

```
# L3 = c; Clear c bytes in m3
```

```
# Loop over the n rows of matrix K
```

lm m2 = [t2]+ r6	# Load m2 = row j of K; t2 +=c
rd1 t4 = m1[t0], 1	# Read t4=RD1(&m1[t0])=X[i][j]; t0+=1
muhu1 m4 = m2, t4	# L4=c;m4=high byteof X[j][0:c]*K[i][j]
satu1 m3 = m3, m4	# L3=c; m3 = saturated add (m3+m4)
bne t0, r5, @loop2	# Branch @loop2 if (j != n)
min1 m3 = m3, r0	# m3 = Reduce 1 to 127 down to 0
sm [t3]+ = m3	# Store P[i][0:c] = m3[0:c]; t3+=c
ibne t5, r4, @loop1	# i++; branch @loop1 if (i != r4)
ret	# Return to caller

The **CONVMNIST** function can be rewritten as two nested loops that do not need to call the **CONVMNIST** function repeatedly. The outer loop1 iterates over all the r rows of X. The @loop1 **lm** instruction loads row i of X into m1 and post-updates t1 to address the next row of A. The mov instruction initializes t2 to r2 (address of matrix K) and the index register t0 to j=0. The clr instruction sets L3=r6=c and clears c bytes of m3 ($c \leq \text{MRL}$).

The inner loop2 loads each row of matrix K into memory region m2, reads one element of m1, and multiplies it by all the elements of m2. It accumulates the product vector in the memory region m3. The @loop2 **lm** instruction loads row j of matrix K from global address t2 into memory region m2. A row of matrix K fit in a memory region ($c \leq \text{MRL}$). The [t2]+ option post-updates the address register t2 to point to the next row in matrix K. The rd1 instruction reads one element (1-bytes) of m1 at index t0=j and transfers its value into t4. The index register t0 is also post-updated.

The **muhu1** (multiply high unsigned) instruction multiplies the unsigned byte elements of **m2** (row i of K) by scalar register **t4= X[j][0:c]** and stores the **high-byte** of the product in **m4**. The low-byte of the product is discarded. For example, if an input byte 135 is multiplied by a weight of 54 then the unsigned product is 7290 and the **high byte** of the product is 28. This is equivalent to multiplying 135 by a weight of 54/256 (equal to 0.21). The integer product is also 28 and the fraction is discarded.

The **satu1** (saturate unsigned) instruction adds the **unsigned bytes** of **m3** and **m4** and saturates the sum at **255**. Saturation is used when a carry is generated, and the unsigned sum is above 255. The **bne** instruction loops back if there are more rows in the weight matrix **K** to multiply-accumulate.

The **min1** instruction, outside the loop, is an activation function with a threshold of **128** (equal to **0.5**). It compares the bytes of **m3** against **0**. If a byte of **m3** is from **1** to **127** then **min1** reduces it to **0**. If a byte is **128** to **255** then **min1** does a **signed** comparison against a negative value from **-128** to **-1** and returns the same value. The output of **min1** is either **0** (if the input is **0** to **127**) or unchanged if **128** to **255**. It implements a **ReLU** activation function with a threshold of **128**.

The **sm8** instruction, outside loop2, stores the accumulated product **m3** in row **i** of **P** at address **t3**. The **[t3]+** option post-updates the address register **t3** to point to the next row in matrix **P**. The **ibne** instruction increments **t5** (**i++**) and repeats loop1 if the incremented **t5** is not equal to **r4** (**r** rows in **X** and **P**) which represent the next image. The **ret** instruction, outside loop1, jumps back to the caller function at the return address in **r31**. The code becomes more compact, knowing that $n \leq \text{MRL}$ and $c \leq \text{MRL}$.

The code above is the same implemented for **CONV1** and **CONV2** for the **MNIST** benchmark except the input size is changed to **12x12x10** for ten images on each has **12x12** obtained after reduction of first max-pooling. The size of the kernel space matrix (**K**) for **CONV2** is also changed into $[(12-5+1)^2 \text{ by } (12)^2] = 64 \times 144 \text{ bytes} = 9216 \text{ bytes} = 9\text{KiB}$ which fitted well in one memory region. The **CIFAR-10** benchmark convolution operation is also implemented in the same way but they a slight difference with the above codes. The **CIFAR-10** each image size is **32x32x3** bytes equal to **3072 bytes = 3KiB**, and for ten images is **30KiB** which still is less than the size of one memory region. Each image has three-pixel matrixes for Red, Green, and Blue, so we loaded thirty

matrices of 32x32 each into one memory region m1 by arranged all three matrices of one image consecutive because it will much easy, to sum up, element-wise of each pixel in RGB into one value. The csum instruction to compute the maximum of three rows **element-wise** gives the result in one matrix of each of the input images. The size of the Kernel space matrix in CIFAR-10 is $(32-5+1)^2 \times (32)^2$ bytes = 802816 byte = 784 KiB which is larger than the one memory region, then we divided the matrix K as shown in Figure 7.3 by allows to load the maximum element to 512 KiB as MRL for single-byte elements.

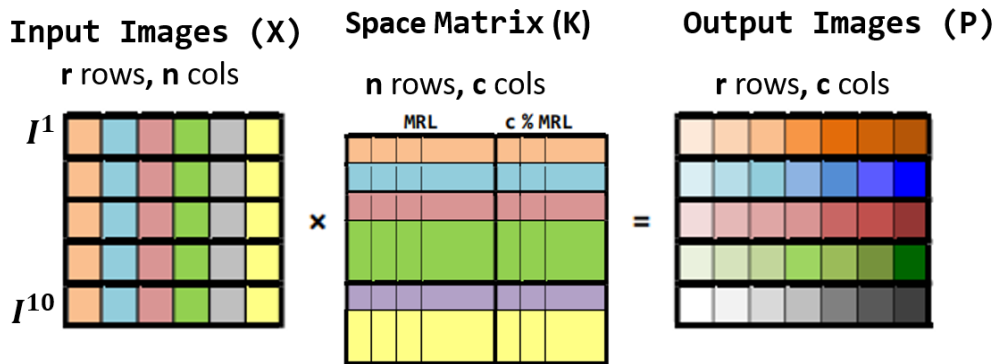


Figure 7. 3 The Convolution Matrixes in CIFAR-10 when $c > \text{MRL}$ in K

```
# CONV CIFAR10: Convolution Matrix Matrix Multiply (1-byte elements)
# r1: Address of matrix X in global memory
# r2: Address of matrix K in global memory
# r3: Address of product matrix P in global memory
# r4: Number r of rows in matrix X = Number of rows in matrix P
# r5: Number n of columns in matrix X = Number of rows in matrix K
# r6: Number c of columns in Matrix K= Number of columns in matrix P
# This version assumes that  $n > \text{MRL}$  and  $c > \text{MRL}$ 
```

```
@CONV CIFAR10
shl r7 = r6, r0
```

```
# CONV CIFAR10 Function
# r7 =  $(c*1)$  bytes per row in K
```

```
@loop0
mov t1, t3 = r1, r3
mov t5 = 0
```

```
# Loop if  $(c > \text{MRL})$  in K
# t1 = address of X, t3 = address of P
# t5 = row index i in X and P (Image)
```

```

@loop1                                # Loop over the I images of matrix X
lm m1 = [t1]+ r5                      # Load m1 = row i of X; t1+=c
mov t2, t0 = r2, r0                  # t2 = address of K, t0 = index j=0,MRL
clr1 m3, t8 = r6                     # t8=L3=MINU(r6,MRL); Clear L3 bytes

@loop2                                # Loop over the n rows of matrix K
lm m2 = [t2]+ r6                      # Load m2 = row j of K; t2 +=c
rd1 t4 = m1[t0], 1                    # Read t4=RD1(&m1[t0])=X[i][j]; t0+=1
muhu1 m4 = m2, t4                     # L4=c;m4=high byteof X[j][0:c]*K[i][j]
satu1 m3 = m3, m4                     # L3=c; m3 = saturated add (m3+m4)
add t2 = t2, r7                       # t2 = &K[i][j:L3]
bne t0, r5, @loop2                   # Branch @loop2 if (j != n)
min1 m3 = m3, r0                      # m3 = Reduce 1 to 127 down to 0
csum m3 = m3, 3                       # Sum of 3 rows element-wise; L3 = L3/3
sm [t3]+ = m3                         # Store P[i][0:c] = m3[0:c]; t3+=c
ibne t5, r4, @loop1                  # i++; branch @loop1 if (i != r4)

shladd r2 = t8, 6, r2                 # r2 = &K[0][j] (j=0,MRL,..)
bnes r6, t8, @loop0                  # branch @loop if (r6 != L3); r6-=L3
ret                                   # Return to caller

```

The @CONV1CIFAR-10 function implements the convolution of Figure 7.3 for CIFAR-10 benchmarks as CONV1 where CONV2 is the same code as the above @CONVMNIST function.

In the @CONV1CIFAR-10 function, the outer loop0 is written to handle a number n of elements of each image in K. If K is greater than MRL, then each iteration of loop0 can load at most MRL bytes of K into m1. It also initializes t0 to address the m1 memory region. The last iteration of loop0 multiplies the last c%MRL elements of each image in X by the last n%MRL rows of K and accumulates the product vector in m3 loop0.

After completing all loop1 iterations, the **sm** instruction stores m3 at the global address of matrix P. The global address in r3 is post-updated to skip MRL elements in P. The **add** instruction advances the global address in r2 to skip MRL columns in matrix K. The last **bnes** repeats loop0

as long as there are more columns in matrix K to multiply and elements of each image in matrix X and matrix P produced.

7.1.2 Pooling Operation

The work was implemented in a 2x2 filter with Stride 2 by using the max pooling operation, which takes a maximum value in the four elements of the filter, as shown in Figure 7.4.

Input Matrix						Max Pooling Output, O		
1	0	0	0	1	0	2	2	1
0	2	0	2	0	1	2	2	1
0	1	1	1	0	0	2	2	2
0	2	1	2	0	1	2	2	2
2	2	2	2	2	0			
0	2	1	2	0	1			

Figure 7. 4 Example of Maximum Pooling Operation

In M-architecture all rows of an image (or multiple images) are in 1D that we loaded them in one memory region, m0. We used clr1 instruction to clear m1 for storing a reduction result. We proposed **cmax** instruction to compute the maximum of **each two elements** in **m0** and write the result in **m1** at the same time when loading which loop1 handles reduction. Then **rmax** instruction is applied to compute the maximum of two rows **element-wise**. Then, the number of elements in **m1** is reduced **by half**. Loop1 handles the loading of the elements that iterates over all the rows of an image or a group of images. We saved back the maximum pooling into global memory by using **sm** instruction. The below codes have been implemented in all operations in the experiments.

```
# r1 : Address of matrix ConvOutput (P) in global memory
# r3 : Number of byte elements in P
# r4: Number of output byte-elements
# r6 : Address of MaxPoolOutput (P) in global Memory
@MaxPool          # MaxPooling Function
clr1 m2 = r4      # L2 = r4; Clear r4 bytes in m2
@loop1           # Loop over the n rows of Matrix P
```

```

lm m0 r2 = [r1]+, r3      # Load m0 = row i of P; r1+=(n*1)
cmax m1 = m0, 2           # m1 = Max(m1[I,j] & m1[I, j-1])
bne r2, r3, @loop1        # t1--, if(t1 != t3) : @loop1
rmax m2 = m1, 2           # m2 = ½(L1) max-2-rows element-wise
sm [r6]+ = m2             # store the ouput of max-pool in global
ret                       # Return to the caller

```

7.1.3 Fully-Connected Layer

The fully connected layer is the classification layer which classifies the images by taking the results of the convolution/pooling process. The fully-connected computation involves forwarding propagation and backpropagation whereby forwarding propagation started from the flattening input and hidden layer as Dense 1, and the second level is between hidden layer and output. In this work, we design ten nodes of neurons in the hidden and output layer but the input nodes depend on the output from extraction processes. Figure 7.5 and Equation 7.1 show the parameters and connection of the fully connected layer.

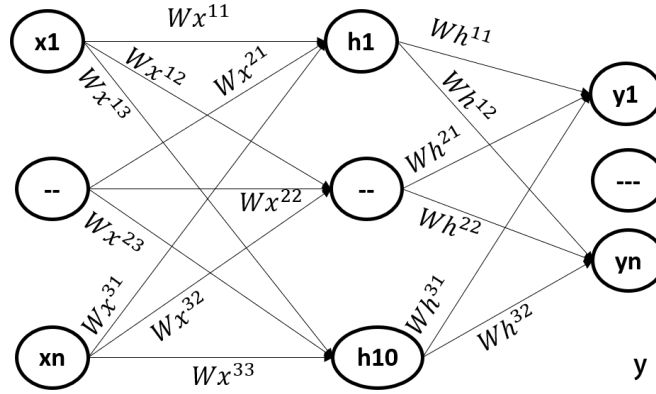


Figure 7.5 Fully-Connected Layer Connection

$$HiddenLayer_j = ReLU(\sum_{i=0}^k (TrainingInput_i HiddenWeight_i + HiddenLayerBias)) \quad (7.1)$$

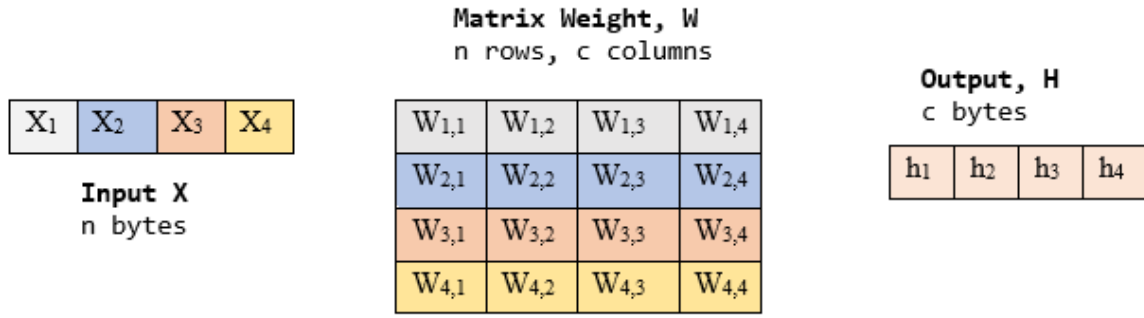


Figure 7. 6 M-Architecture Implementation of forwarding Propagation

```

H[0:c] = 0 // Clear all c elements of H

for i in 0 to n-1 // Repeat over all n rows of W

    P[0:c] += (X[i] * W[i][0:c]) + b[i] // Multiply X[i] by row W[i][0:c]

```

In M-architecture, we implemented the forward propagation of each layer by using a vector to matrix multiplication. The following **FCLayer** function multiplies an input vector **X** of length **n** bytes by a matrix **W** with **n** rows and **c** columns to produce an output product vector **H** of length **c** bytes.

The inputs to a hidden layer are **signed bytes** that carry positive values from **0** to **+127** and the edges of a hidden layer carry **signed weights** stored in a matrix of **signed bytes** from **128** to **+127** then the neurons at the outputs of a hidden layer compute the weighted average of their inputs using vector-matrix multiplication that saturates at **128** and **+127**. The saturated weighted averages pass through a **ReLU** activation function that modifies all negative values to **0**. The output of the activation function is positive values from **0** to **+127**.

The first **lm** instruction loads vector **X** into **m1**. Vector **X** is fit in **m1** ($n \leq \text{MRL}$). The **mov** instruction initializes **t2** = **r2** = address of matrix **W** and the index register **t0** = **0**. The **clr1** instruction sets **L3=r5=c** and clears **c** bytes of **m3**, assuming $c \leq \text{MRL}$.

A loop is written to load each row of matrix **W** into memory region **m2**, multiply this row by one element of vector **X**, and accumulate the product vector in memory region **m3**. The **@loop lm** instruction loads row **i** of matrix **W** from global address **t2** into memory region **m2**. A row of matrix **W** fit in a memory region ($c \leq \text{MRL}$). The **[t2]+** option post-updates the address register **t2** to point to the next row in matrix **W**.

The **rd1** instruction reads a **one-byte** element of **m1** at index **t0=i** and transfers its value into **t1**. Register **t0** is also post-updated to index the next byte in **m1**. The **muh1** (**m**ultiply **h**igh) instruction multiplies the signed byte elements of **m2** (row **i** of **W**) by scalar register **t1=X[i]** and stores the **high-byte** of the product in **m3**. The low-byte of the product is discarded. For example, if an input byte **65** is multiplied by a weight of **-34** then the signed product is **2210** and the **high byte** of the product is **-9**. This is equivalent to multiplying **65** by weight of **34/256** (equal to **0.13**). The integer product using the floor function is **-9**.

The **sat1** (**s**aturate) instruction adds the **signed bytes** of **m3** and **m4** and saturates the sum at either **128** or **+127**. Saturation is used when there is overflow and the sum is out of range. The **max1** instruction, outside the loop, is an activation function with a threshold of **0**. It compares the bytes of **m3** against **0**. If a byte of **m3** is **negative** from **-128** to **-1** then **max1** forces it to become **0**. It implements a **ReLU** activation function with a threshold of **0**.

The **sm** instruction, outside the loop, stores the accumulated and rectified product in **m3** in vector **H** at global address **r3**. The **ret** instruction returns back to the caller function.

```

# FCLayer: Fully Connected with Rectified Output (Byte elements)
# r1, r2, r3: Global address of X, W, H
# r4: Length n (bytes) of the input vector X = Number of rows in W
# r5:Length c (bytes) of the output vectorH = Number of columns in W
# This version assumes that n and c are less than or equal to MRL

```

<pre> @FCLayer lm m1 = [r1], r4 mov t2, t0 = r2, r0 clr1 m3 = r5 @loop lm m2 = [t2]+ r5 rd1 t1 = m1[t0], 1 muh1 m4 = m2, t1 sat1 m3 = m3, m4 bne t0, r4, @loop max1 m3 = m3, r0 sm [r3] = m3 ret </pre>	<pre> # FCLayer Function # L1=n; Load m1 = vector X # t2 = address of W, t0 = index i # L3=c; Clear c bytes in m3 # Loop over the n rows of matrix W # L2=c; Load m2 = row W[i]; t2+=c # Read t1=RD1(&m1[t0])=X[i]; t0+=1 # L4=c; m4 = high byte of m2*X[i] # L3=c; m3 = saturated add (m3+m4) # branch @loop if (i != n) # m3 = negative bytes are made 0 # Store vector H = m3 # Return to caller </pre>
---	--

7.1.4 SoftMax Operation

The SoftMax implemented in the work to convert the output of a fully connected layer into probability so that we can evaluate the errors of each label. The SoftMax Function stated with initializing r2 to 0 and the loop starts to load the output vector (y) into m0 with post-updated r1++ to load more elements. We propose **cexp1** instruction computes the exponential value of each element in m0, and convert the result elements into byte integers. The **rsum** instruction is applied to accumulates the sum of the result into r2. The **divl1** instruction computes the low n-byte of the division between the n-byte elements in m1 and accumulated result in r2. The **bne** instruction is checking if all elements are finished loading otherwise the loop repeats until finished loaded all elements. Then, **sm** instruction, outside the loop, stores the low n-byte of the division from m2 at global address **r5**. The **ret** instruction returns back to the caller function.

```

# r1 = Address of output array in global memory
# r2 = Holds the sum of the output
# r3 = Count n of 1-byte elements in y as output
@Softmax          # SoftMax Function
mov r2 = r0        # Initialize: r2 = 0
@loop
lm m0, r5 = [r1]+, r3  # L0=..; m0=LM(r1,L0)=y'; r1+=L0 r5=L0
cexp1 m1 = m0         # exponents m1 =  $e^{mb}$ single-bytes elements
rsum r2 = m1, r2      # R=RSUM(m1,L1); r2=rSUM(R,r2)
divl1 m2 = m1, r2     # M2 is the integer low n-byte result
bne r3, r5, @loop     # branch if (r3!=L0); r3-=L0
sm [r5]+ = m2        # Store P[y'] = m2[i];
ret                  # return to the call

```

7.1.5 Updating Weights

Backpropagation is finding the best weight values of the labels by adjusting the old value weight due to error values of each node, and this process is known as training. Figure 7.7 shows the structure and analysis of the updating weight. The expansion formula in Figure 7.7 is the combination of Equation 2.7 and Equation 2.11 that described the weight computation. The vector values obtained from the SoftMax operation represent the computed output (y) and the capital Y is the desired output which has a value of 1 equal to 100%. This experiment implemented the value of Y equal to 100 because the computed output y is in byte integers as explained in the SoftMax operation.

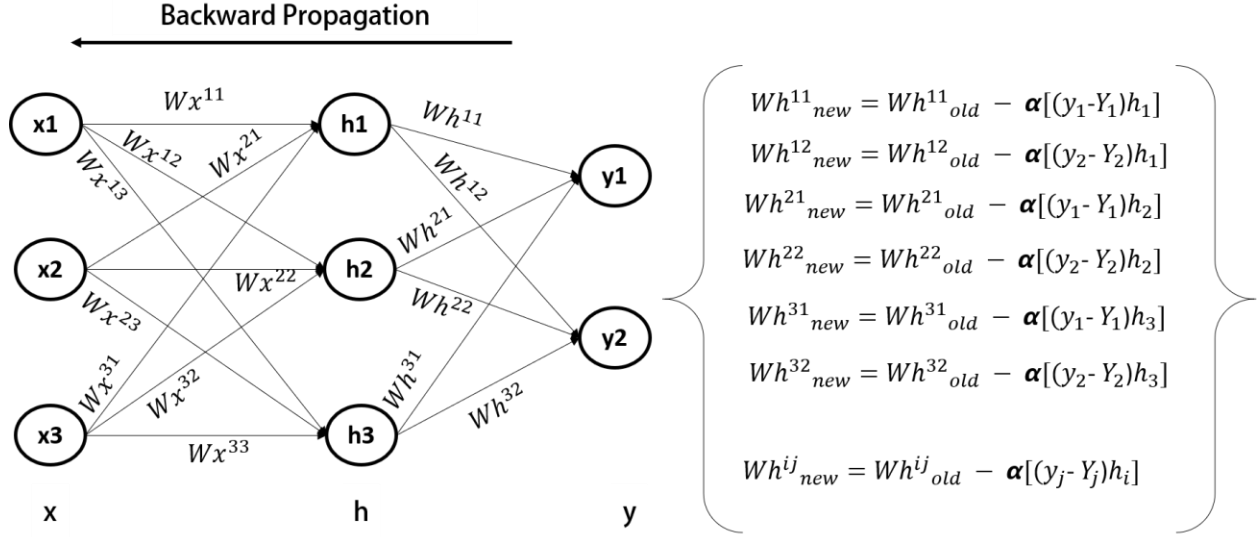


Figure 7. 7 Updating Weights Analysis

Wh_{old}^{11}	Wh_{old}^{12}
Wh_{old}^{21}	Wh_{old}^{22}
Wh_{old}^{31}	Wh_{old}^{32}

-

$\alpha(y_1 - Y_1) h_1$	$\alpha(y_2 - Y_2) h_1$
$\alpha(y_1 - Y_1) h_2$	$\alpha(y_2 - Y_2) h_2$
$\alpha(y_1 - Y_1) h_3$	$\alpha(y_2 - Y_2) h_3$

=

Wh_{new}^{11}	Wh_{new}^{12}
Wh_{new}^{21}	Wh_{new}^{22}
Wh_{new}^{31}	Wh_{new}^{32}

Figure 7. 8 Updating Weights Implementation

The formula in Figure 7.6 obtained after deriving from Equation 2.11 and Error Equation as following:

$$Wh_{new}^{ij} = Wh_{old}^{ij} - \alpha \frac{\partial E}{\partial W^{ij}} \quad \text{and} \quad E = \frac{1}{2} (target - actual)^2$$

According to the chain rule, the derivative of the error respects to the weight will be:

$$\frac{\partial E}{\partial W^{ij}} = \frac{\partial E}{\partial target} \times \frac{\partial target}{\partial W^{ij}} \quad (7.1)$$

$$target(y_1) = h_1 Wh^{11} + h_2 Wh^{21} + h_3 Wh^{31} \quad (7.2)$$

Now, we substitute Equation 7.2 and Error formula into Equation 7.1 for the change of error with respect to the weight value of W^{11} as reference.

$$\frac{\partial E}{\partial W^{ij}} = \frac{\partial [\frac{1}{2} (target - actual)^2]}{\partial target} \times \frac{\partial target}{\partial W^{ij}}$$

$$\frac{\partial E}{\partial W^{11}} = \frac{\partial [\frac{1}{2} (y_1 - Y_1)^2]}{\partial y_1} \times \frac{\partial (h_1 W^{11} + h_2 W^{21} + h_3 W^{31})}{\partial W^{11}}$$

$$\frac{\partial E}{\partial W^{11}} = 2 * \frac{1}{2} (y_1 - Y_1) \frac{\partial (y_1 - Y_1)}{\partial y_1} \times h_1$$

$$\frac{\partial E}{\partial W^{11}} = (y_1 - Y_1) \times h_1$$

Therefore, the updated weights will be:

$$Wh_{new}^{ij} = Wh_{old}^{ij} - \alpha [(y_1 - Y_1) \times h_1] \quad (7.3)$$

To gain data parallelism, we implemented the updating weights in the M-architecture name as **UpdatingWeights** Function which started by initialized the local memory index t0, and t1, then we clear m5 and m7 that can store the difference between computed output vector and the actual output vector. Then we loaded the computed output vector into m0, then the sub1 instruction computes the difference between vector y and actual output (Y) into m2. The second lm instruction loads the input vector h into m3. The loop initiated to handles the loading of the old weight matrix value into m3. The rd1 instruction reads one element (1-bytes) of m3 at index t0=j and transfers its value into t1. The index register t0 is also post-updated. The **muh1** (**m**ultiply **h**igh) instruction multiplies the signed byte elements of **m0** (y[i]-Y) by **t1=h[i]** and stores the **high-byte** of the product in **m4**. The low-byte of the product is discarded. The **sat1** (**s**aturate) instruction adds the

signed bytes of **m5** and **m4** and saturates the sum at either **128** or **+127**. Saturation is used when there is overflow and the sum out of range. The **bne** instruction loops back if there are more images in the input matrix **X** to multiply-accumulate. The second **muh1** (**m**ultiply **h**igh) instruction multiplies the signed byte elements of **m5** $((y[i]-Y)*h[i])$ by scalar register **t2=α** and stores the **high-byte** of the product in **m5**. The low-byte of the product is discarded.

The **lm** instruction handles the loading of the old weight matrix value into **m6** but at the same time, we subtract the old weight matrix with **m5** saved into **m7** after **sub1** instruction is applied. The **sub1** instruction subtracts and saturates the result at either **128** or **+127**. Then, **sm** instruction, outside the loop, stores the low n-byte of the W_{new} from **m7** at global address **r7**. The **ret** instruction returns back to the caller function.

```
# r1, r2, r6: Global address of Y, y, h
# r3: global address of matrix value for old weights (Wold)
# r7: Global address of Matrix of Wnew
# t2: learning rate value register α = 0.01 (~1)
# r5: Length c (bytes) of the Weights (Wold & Wnew)
# r4: Length n (bytes) of the vector y & Y
# r8: Length n (bytes) of the vector h

@UpdatingWeights      # start UpdatingWeights
mov t0, t1 = r0, r0   # t1 = 0, t0 index i = 0
clr1 m5 = r5          # Clear c bytes in m5
clr1 m7 = r5          # Clear c bytes in m7
lm m0 = [r2]+ r4      # Load m1 = computed output, y; L0=r4=n;
lm m1 = [r1]+ r4      # Load m1 = actual output, Y; L1=n;
sub1 m2 = m0 m1       # m2[i] = y[i] - Y[i];
lm m3 = [r6]+ r8      # Load m3 = Input vector h; L2=r8
@loop                # Loop start to compute
rd1 t1 = m3[t0] 1     # Read t1=RD1(&m3[t0])=h[i]; t0+=1
muh1 m4 = m2 t1       # multiplying sign elements-wise (y[i]-Y[i])*h[i]
sat1 m5 = m5 m4       # m5 = saturated add (m5+m4); if(L5 > L4: i++)
bne t0 r8 @loop      # branch if (t0 != r8);
muh1 m5 = m5 t2       # m5 = α((y[i]-Y[i])*h[i])
```

```

lm m6 = [r3] r5      # Load m6 = Old Weight, Wold; L6=r5;
sub1 m7 = m6 m5      # m7[i] = Wold[j][0:c] -  $\alpha((y[i]-Y)*h[i])$ 
sm [r7] = m7         # Store new weight vector = m7
ret                  # Return to caller

```

7.2 Recurrent Neural Networks

In this work, the RNN algorithms shown in Figure 6.10 were implemented using the word2vec and 1-Hot representation methods. The experiments compute the similarity between three sentences by encoding with the mentioned two methods. The implementation of both methods is the same except that the input of 1-Hot encoding provides one integer value for each word while word2vec generates many integers for each word, but in this work, we generated vectors for each word in the sentences to test.

The RNNFunction is implemented by combining all same vectors (i.e., X, U, W for all sentences) into one memory region to benefit parallelism. The function starts by initialized all local index t0, t1, t2, and t3 that we used to index position in memory regions. Then we cleared all single byte in the memory region m5 for saving the result, then we initialize loop1 to load the input vector X which handles the loading between vector X, U, and W into memory region m0, m1, and m2 respectively. Then multiple the elements of vector X and U element-wise by using muh1 instruction and the result recorded in m3. The low-byte of the product is discarded. Loop2 starts to compute the output H at time t by using madd instruction after reading values of node weight into t0 and m3 into t1 by using rd1 instruction. The result is pushed as input to the second node for the next word and saved into m5. The bne instruction repeats the computation at a time increasing until all words are finished in the sentence. The max1 instruction, outside the loop, is an activation function with a threshold of 0. It compares the bytes of m5 against 0. If a byte of m5 is negative from -128 to -1 then max1 forces it to become 0. Then sm instruction writes the result in m5 into

global memory address through r6 and calling the return function. Therefore, we computed three sentences at one program as data parallelism as more details in the following RNNFunction Instructions.

```
# r1: Address of Sentences matrix Input (X) in global memory
# r2: Global address of weight Matrix (U)
# r3: Global address of weight Matrix (W)
# r6: Address of output matrix (Y) in global Memory
# r5: Number of Column in Matrix A
# t3: output at time h(t-1), start t3 =0 as initial value
# t4: number of cells in RNNs design (t4=5)
```

<pre>@RNNFunction mov t0, t1 = 0, 0 mov t2, t3 = 0, 0 clr1 m5 = 0 @loop1 lm m0, r4 = [r1]+, r5 lm m1 = [r2]+, r5 lm m2 = [r3]+, r5 muh1 m3 = m0, m1 @Loop2 rd1 t0 = m2[t2], 1 rd1 t1 = m3[t4], 1 madd t3 = t3, t0, t1 wr1 m5[t5],1= t3 bne t2, r4, @Loop2 bne r4, r1, @Loop1 max1 m5 = m5, r0 sm [r6]+ = m5 ret</pre>	<pre>#RNN Function # t0 = 0, t1 =0 # t2 = local index for W, t3 = 0 # clear m5 =0 # Loop over the n rows of Matrix X # Load m0 = row i of X; r1+=c # Load m1 = row i of U; r2+=c # Load m2 = row i of W; r3+=c # m3[i] = X[i]U[i] # start compute the output # Read t5=RD1(&m2[t2*1])=W[i]; t2+=c # t6=RD1(&m3[t4])=X[i]U[i];t4+=c # t3[i] = (t3[t-1]* W[i]) + X[i]U[i] # m4[i] = t3, t3 +=1 # if(t2 !=t4) : not complete cells # if (r4 != t0): not finished loaded # m3 = negative bytes are made 0 # store the output (Y) in global # Return to the caller</pre>
---	---

7.3 Performance Evaluation

In this section, we discussed the analysis of benchmarks in terms of performance and functionality by our M simulator. The simulator gives an estimated performance result of the number of

operations, number of instructions, number of memory and scalar instructions, number of cycles, Cycle Per Instructions (CPI), and average number of operations per cycle (ANPC) when we run the benchmarks. The output values are visible in both local memory and global memory, which makes it easy to check the result of the operation. Therefore, the simulator helps us to capture the performance efficiency of our benchmark in SPM. We have recorded the results of our benchmark estimation as follows:

7.3.1 Results and Discussions

We simulate the CNN and RNN algorithms in the M simulator to record the performance matrices. For the CNN algorithms, we simulate one part at a time until the last step of the algorithm for each benchmark (i.e., MNIST and CIFAR-10). We train the algorithm by loading ten images from each benchmark to achieve data parallelism in our M architecture. The main goal of training is to obtain a certain value (label) for each image that can help in classification during the testing or validation process. The number of operations per cycle as an estimation result was recorded in Table 7.1 and Table 7.2, respectively. In the experiment, each operation was performed in 1 as NS1, 4 as NS4, 8 as NS8, 16 as NS16, 32 as NS32, and 64 as NS64 number of execution units.

Table 7. 1 Number of Operation Per Cycles in MNIST Benchmark

Operation	MNIST Benchmark: Number of Operations Per Cycles						
	Data Set Size	NS1	NS4	NS8	NS16	NS32	NS64
Convolution 1 + ReLU	Images: 28x28x10 Kernel: 576x784	1.999	7.999	15.998	31.993	63.974	127.895
Max-Pooling 1	24x24x10	3.276	12.191	21.838	38.110	59.024	80.561
Convolution 2 + ReLU	Images: 12x12x10 Kernel: 64x144	1.998	7.979	15.909	31.680	62.736	122.969
Max-Pooling 2	8x8x10	0.709	2.436	3.946	6.233	8.421	10.088
Dense 1 Layer + ReLU	160x10	0.585	2.331	4.634	9.192	18.038	34.731
Dense 2 Layer + ReLU	10x10	2.941	9.090	13.186	19.047	23.300	25.876
SoftMax	10	0.483	1.304	1.690	2.264	2.580	2.735
Updating Weights	10x10	5.228	18.149	29.747	47.492	65.009	78.757

Figure 7.9 shows that the increasing number of execution units helps to perform a large number of operations in convolutions (CONV1 & CONV2) compared to other parts, which is due to the large source data involving matrix multiplication between kernel space matrix and image size. However, overall, we show that increasing the execution unit for all layers results in more operations being executed in parallel.

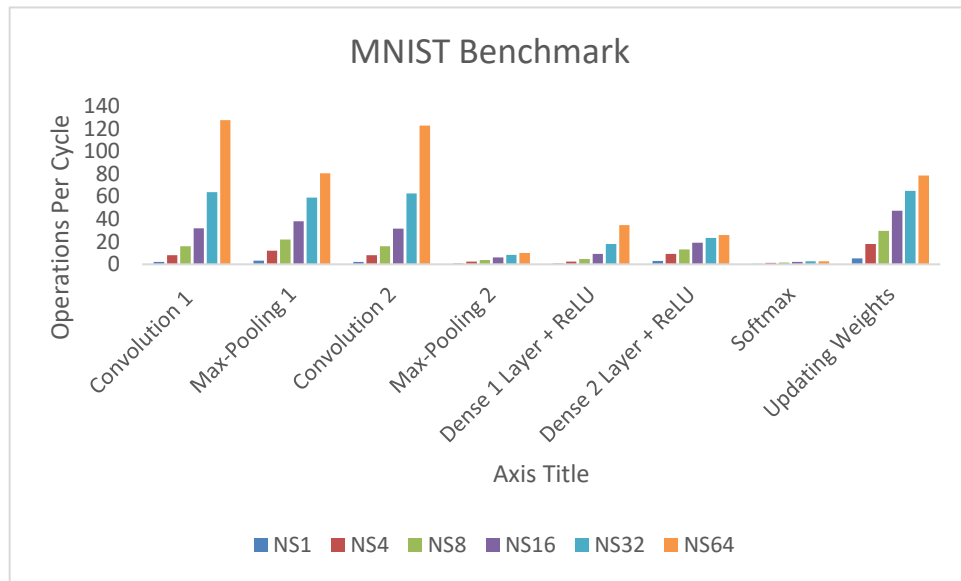


Figure 7.9 MNIST Benchmark CNNs (Operations Per Cycles)

The Max-Pooling 2 is the small number of operations per cycle compare with the Max-Pooling 1 due to the reduction of the input size. We calculate the speedup of each layer by taking NS1 as the baseline and comparing it with other execution units. The dense 1 layer provides almost the same speedup value as convolution layers despite it has a small number of operations per cycle than convolution operations. The computed estimation result is shown in Figure 7.10.

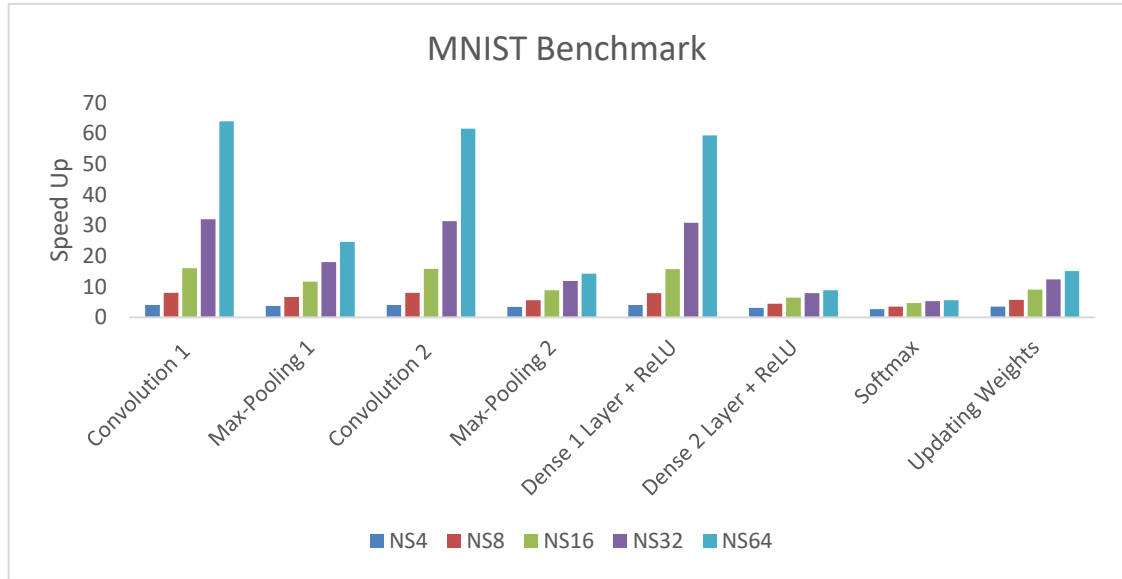


Figure 7. 10 MNIST Speedup Normalize to Single Execution Unit

The performance of the CIFAR-10 benchmark is also recorded in the experiments as shown in Table 7.2 the number of operations per cycle for each layer in CNNs algorithms.

Table 7. 2 Number of Operation Per Cycles in CIFAR-10 Benchmark

Operation	CIFAR-10 Benchmark: Number of Operations per Cycles						
	Data Set Size	NS1	NS4	NS8	NS16	NS32	NS64
Convolution 1 + ReLU	Images: 32x32x3X10 Kernel: 784x1024	4.499	17.999	35.999	71.999	143.999	287.998
Max-Pooling 1	28x28x10	2.944	11.157	20.851	36.865	59.847	86.950
Convolution 2 + ReLU	Images: 14x14x10 Kernel: 100x196	2.999	11.996	23.985	47.943	95.773	191.095
Max-Pooling 2	10x10x10	2.613	7.537	10.989	14.251	16.736	18.334
Dense 1 Layer + ReLU	250x10	0.590	2.355	4.696	9.334	18.440	35.995
Dense 2 Layer + ReLU	10x10	2.941	9.090	13.953	19.047	23.300	26.229
SoftMax	10	0.483	1.304	1.818	2.264	2.580	2.774
Updating Weights	10x10	5.228	18.147	30.862	47.492	65.009	79.709

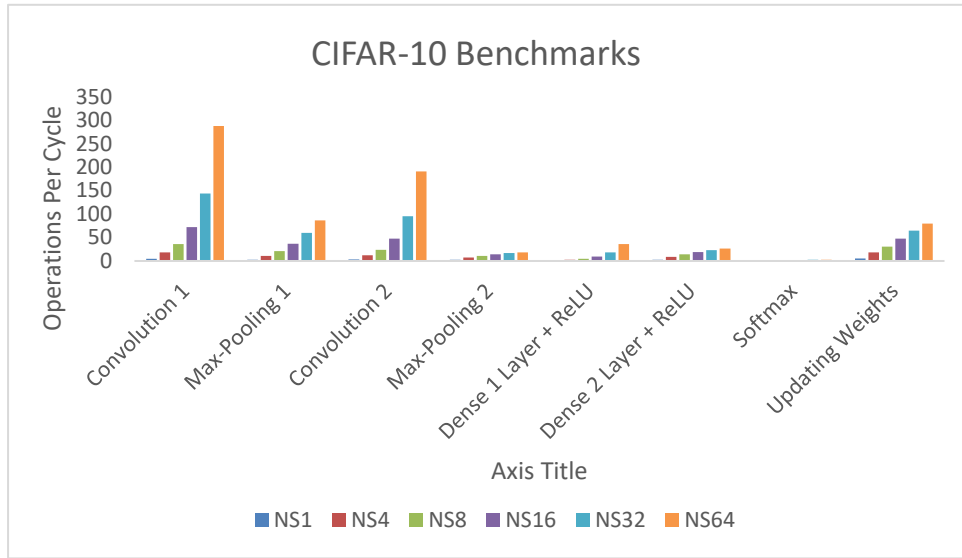


Figure 7. 11 Number of Operations Per Cycles in CIFAR-10 Benchmark

The estimated number of operations per cycle in CIFAR-10 increases as the number of slices increases, which is due to an increasing number of execution units. Although the CIFAR-10 benchmark is larger than MNIST in terms of data size, the performance in the M architecture shows little difference between them. We also calculated the speedup of the CIFAR-10 benchmark by normalizing with a single execution unit, as shown in Figure 7.12

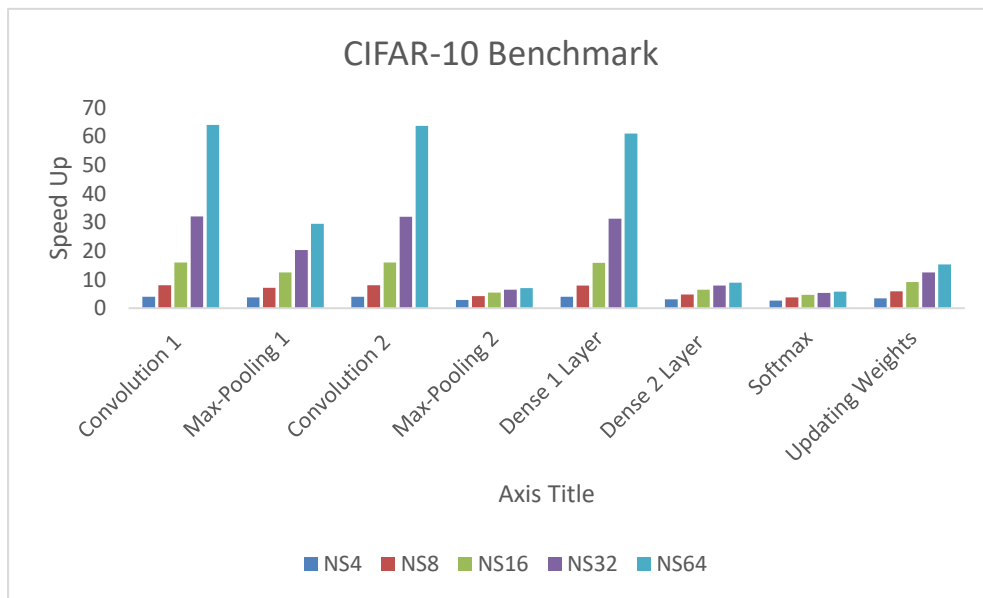


Figure 7. 12 Speedup Normalize to Single Execution Unit (CIFAR-10 Benchmark)

In general, data parallelism computation in the M architecture is very beneficial when the source is big data, and it can significantly improve performance by using a larger number of execution units. The fully connected architecture can also be beneficial when we use multiple neurons, especially in the Dense-2 layer, which does not contribute much to the computation due to the small number of nodes chosen to analyze our proposed instructions for deep neural networks.

The experiment was repeated ten times in the fully connected layer by using backpropagation to adjust the weight and improve the performance. We computed SoftMax by taking the reminder integer of each label ten times to get the label value of the image so that it can be used during testing and validation. Table 7.3 shows the obtained SoftMax output during backpropagation in the CIFAR-10 benchmark.

Table 7.3 Training Output for the Labels in CIFAR-10

Iteration	CIFAR-10 Benchmark Labels										
	Airplane	Automobile	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Trunk	Total
1	6	5	1	14	16	12	8	16	10	8	96
2	6	1	19	0	7	11	14	8	9	22	97
3	8	11	2	7	7	14	10	14	13	10	96
4	2	13	10	17	10	11	2	13	11	7	96
5	11	16	16	6	7	7	3	14	6	8	94
6	0	18	17	1	16	12	1	10	5	15	95
7	0	7	23	10	13	15	2	10	0	14	94
8	14	13	21	9	4	16	1	0	14	1	93
9	24	5	17	10	1	12	3	16	1	5	94
10	18	18	10	3	19	4	8	8	3	5	96
AVG	8.9	10.7	13.6	7.7	10	11.4	5.2	10.9	7.2	7.4	

The process of quantizing the filter byte elements into integers reduces the accuracy of the training, as shown in the result Table 7.3, where the total value is obtained by summing the values of each label. The average value is obtained by dividing the obtained values by the number of iterations

(i.e., 10). The average value can be used to identify the label in the dataset during the test experiment. We visualize the average values of the CIFAR-10 labels in Figure 7.13.

Table 7.4 shows the obtained simulation output of the SoftMax operation in the MNIST benchmarks as follows:

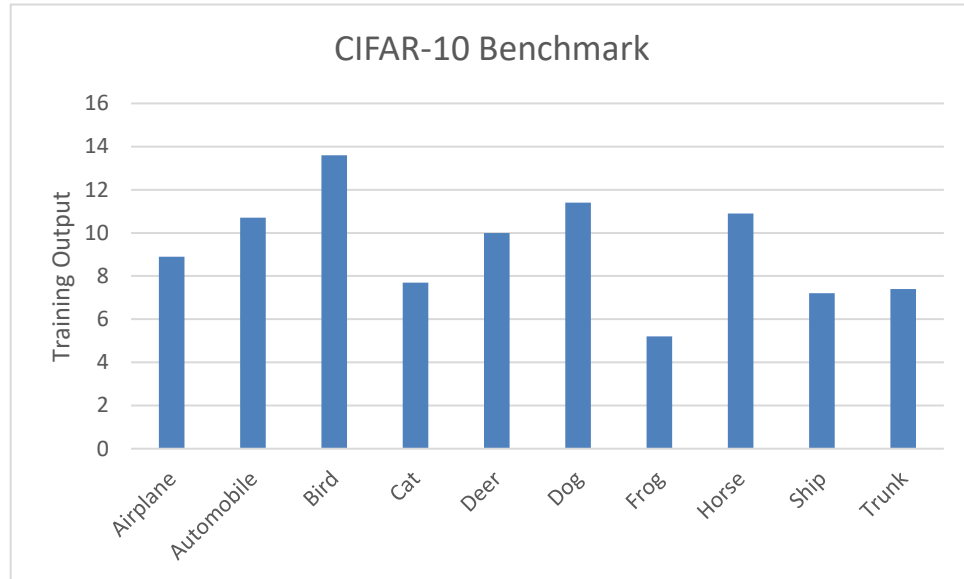


Figure 7. 13 The Obtained Estimated Label Values of CIFAR-10 Benchmark

Table 7. 4 Training Output for the Labels in MNIST-10

Iteration	MNIST Benchmark Label										Total
	0	1	2	3	4	5	6	7	8	9	
1	12	5	18	13	3	0	10	13	6	15	95
2	17	9	0	9	8	6	17	11	8	11	96
3	5	5	8	17	7	10	17	4	3	19	95
4	13	14	14	14	0	7	13	1	7	11	94
5	13	2	8	1	17	15	7	8	9	15	95
6	12	12	10	24	0	3	4	10	12	6	93
7	5	14	15	2	5	9	15	8	12	10	95
8	13	16	7	16	4	11	12	0	9	7	95
9	15	5	3	15	18	1	9	7	12	10	95
10	16	17	17	10	11	2	13	0	0	9	95
AVG	12.1	9.9	10	12.1	7.3	6.4	11.7	6.2	7.8	11.3	

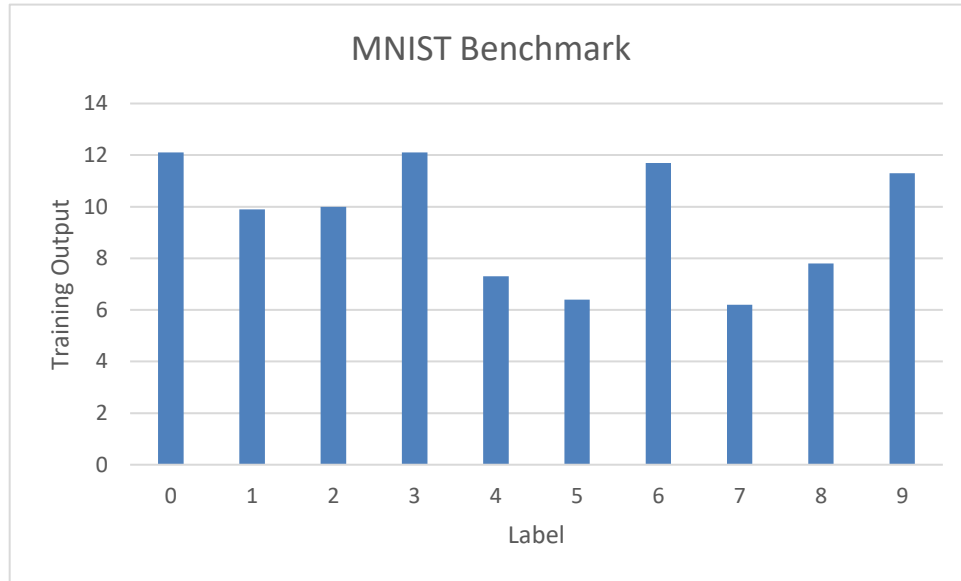


Figure 7. 14 The Obtained Estimated Label Values of CIFAR-10 Benchmark

Figure 7.14, the obtained label values in the MNIST benchmark can be as difficult to classify between labels 0 and 3 in the same way as labels 5 and 7. This means that in future work, the training experiments will need a larger number of iterations (epochs) that can provide a unique value of the labels.

We also recorded the number of operations per cycle in the RNNs algorithm by simulating the three sentences after encoding them into the 1-hot representation and the word2vec word embedding method. Table 7.5 shows the obtained estimated performance of the RNNs algorithms.

Table 7. 5 Number of Operation Per Cycles in RNNs

Encoding Method	Dataset Size	Number of Operation Per Cycles					
		NS1	NS4	NS8	NS16	NS32	NS64
1 Hot Encoding	5x3x	4.891	9.137	10.682	11.669	12.234	12.538
Word2Vec	5x3x20	28.391	32.816	33.692	34.147	34.379	34.497

The result in Table 7.5 shows that word2vec encoding is better than 1-Hot encoding, which is because word2vec represents each word in many vectors, while 1-Hot represents one word in one vector. However, both methods do not improve significantly when the number of execution units is increased because RNNs are sequential over time t and cannot be executed in a fully data-parallel manner. The obtained result is visualized in Figure 7.15, which shows the number of operations per cycle.

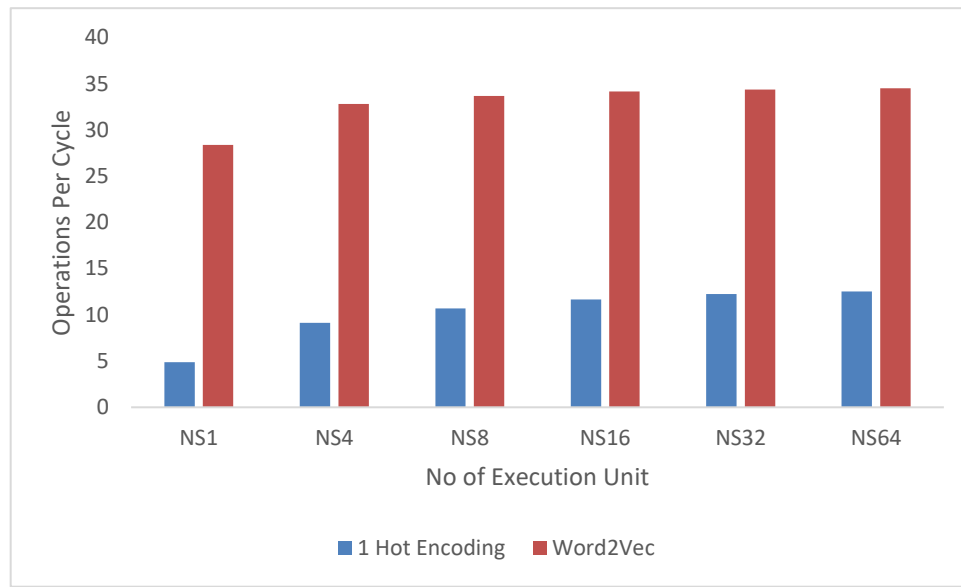


Figure 7. 15 RNN Encoding Methods (Number of Operations Per Cycle)

Figure 7.16 shows the speedup variation between 1-hot method and word2vec method which was obtained by normalized to baseline NS1.

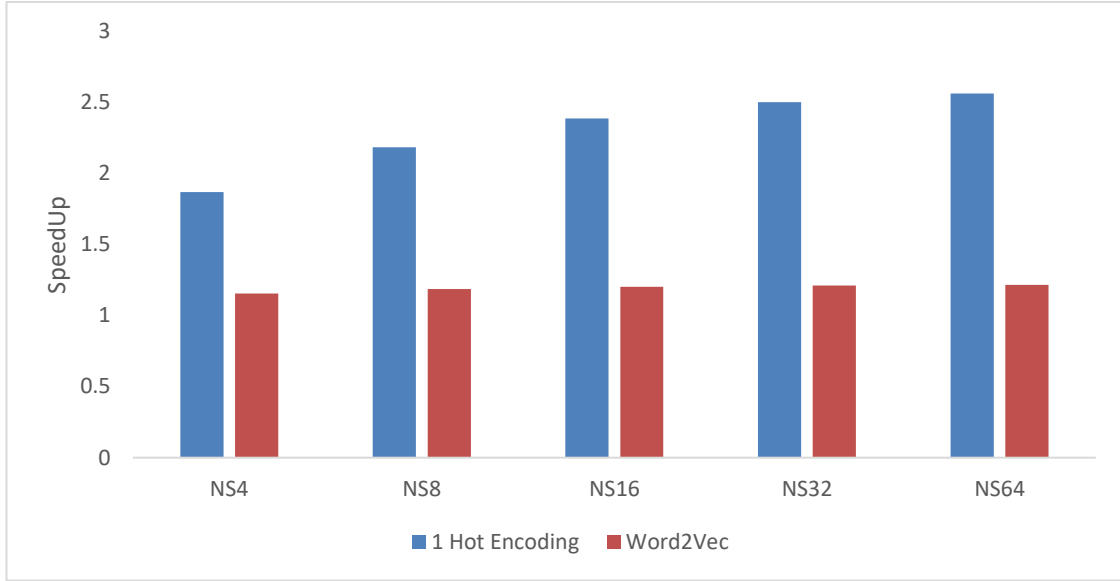


Figure 7. 16 RNN Encoding Methods (Speedup)

We also evaluated the performance of the M-Architecture instruction set by comparing it with two accelerators based on the RISC-V architecture instruction set. We compared the clock cycle performance of the four learning operations (i.e., convolution, ReLU, pooling, and matrix addition). The author [94] presented the better performance of the Coprocessor accelerator compare with the RV32IM accelerator. However, the M-architecture instruction set performed fewer clock cycles than the instruction set proposed by the author [94]. This due to its features of loading a block of data in one instruction, and computes the operation in local memory. Table 7.6 shows the performance of clock cycles of the three type of the instruction set as follows:

Table 7. 6 Number of clock cycles of the operation implemented by different instructions

Algorithm	RISC-V RV32IM Instruction [94]	RISC-V Coprocessor Instruction [94]	M-Architecture Instruction
Convolution (4x4 matrix, 3x3 Kernel)	12982	2070	376
Pooling (6x6 matrix and 2x2 filter)	2452	1706	31
ReLU (6x6 matrix)	1249	1101	12
Matrix Addition (6x6 matrix)	2755	2005	23

The M-simulator used to estimate the performance of clock cycles in M-Architecture instruction is shown in Figure 6.19 and Figure 6.22. Figure 7.17 shows the acceleration ratio by dividing the performance clock cycles of the accelerators based on RISC-V instruction with the performance clock cycle in M-Architecture Instruction. The ratio values represent how many numbers of clock cycles were reduced by M-architecture instruction for each operation compared with Coprocessor and Rv32IM Instruction.

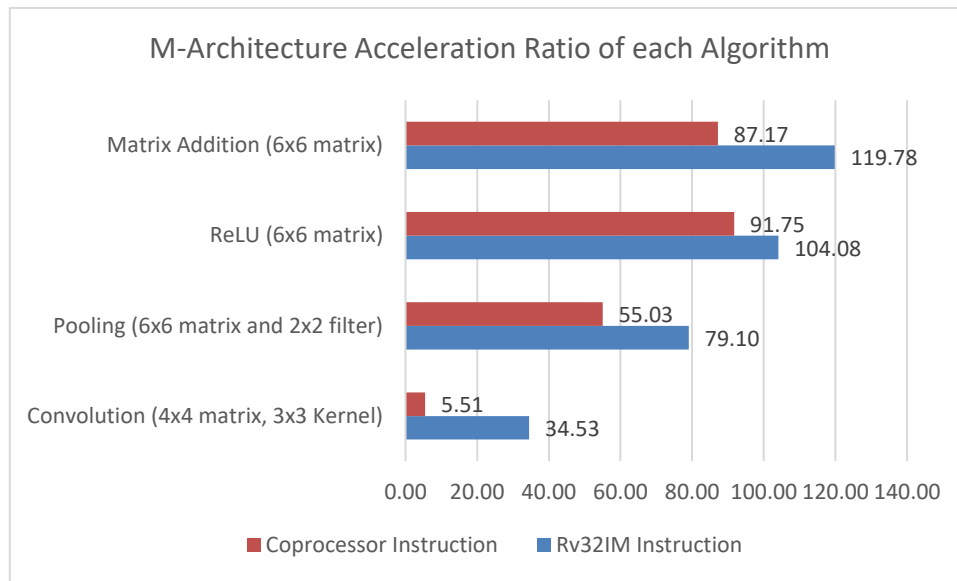


Figure 7. 17 M-Architecture Acceleration Ratio of the CNN Operations

CHAPTER 8

CONCLUSION AND FUTURE WORK

The implementation of DNN algorithms required powerful machines or special accelerators that can support the workload due to the complex structure with many operations and big data. Moreover, existing general-purpose processors are not suitable for implementing DNN algorithms in terms of performance and power consumption. Several architectures (e.g., LeNet, GoogLeNet, VGGNet, ZfNet, AlexNet) have been proposed to overcome the difficulty of DNN acceleration in hardware, but are still not flexible enough to be a standard.

In this research work, we have studied the implementation of deep neural networks (DNN) using Software Programmable Memory (SPM) which is capable of accelerating the computations in local memory. The SPM is divided into regions M0, M1,...M15 where each region can hold a large vector and vector execution unit (ExU). This helps to process data in parallel, reduce access time and increase overall performance. Our general purpose instructions like muhu1, cmas, rmax, min1, satu1, lm1, sm1, bnes were used to compute the deep neural network algorithms (CNN and RNN) in inflexible and better performance.

M-simulator was used to check the functionality and performance evaluation of the proposed algorithms. The number of operations, the number of cycles, and the average number of operations per cycle were recorded while conducting the experiment. The results show that the algorithms work well in our M architecture. It was also observed in the work that the estimated speedup leads to better performance, especially when the number of executions increases. Another reason that improves the performance of our work is the quantization of all elements in a 1-byte integer, but

it decreases the training accuracy of the result due to neglecting some data during computation as shown in our result. However, we managed to calculate the number of operations per cycle and their speedup values of deep neural network algorithms by using our proposed general-purpose instruction set in M architecture. MNIST and CIFAR-10 benchmarks are used to analyze the CNN algorithms, while for RNNs algorithms, the 1-hot and word2vec coded methods have been implemented.

In future work, the work can be extended to implement a large number of neurons in a fully connected layer so that a better value for labelling accuracy can be achieved. The number of iterations should be increased during training until a unique value for each label is achieved so that images can be easily classified during testing or validation. The work can also be extended for training large images such as CIFAR-100, ImageNet, and other Natural Language Processing Applications.

References

- [1] A. G. Ivakhnenko and V. G. Lapa, “Cybernetics and forecasting techniques,” 1967.
- [2] K. Fukushima and S. Miyake, “Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition,” in *Competition and cooperation in neural nets*, Springer, 1982, pp. 267–285.
- [3] A. G. Ivakhnenko, “Polynomial theory of complex systems,” *IEEE Trans. Syst. Man. Cybern.*, no. 4, pp. 364–378, 1971.
- [4] J. Schmidhuber, “Deep learning,” *Scholarpedia*, vol. 10, no. 11, p. 32832, 2015.
- [5] R. Dechter, “Learning while searching in constraint-satisfaction problems,” 1986.
- [6] I. Aizenberg, N. N. Aizenberg, and J. P. L. Vandewalle, *Multi-valued and universal binary neurons: Theory, learning and applications*. Springer Science & Business Media, 2013.
- [7] D. Ciregan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in *2012 IEEE conference on computer vision and pattern recognition*, 2012, pp. 3642–3649.
- [8] B. Agüera y Arcas, “Art in the age of machine intelligence,” in *Arts*, 2017, vol. 6, no. 4, p. 18.
- [9] G. W. Smith and F. F. Leymarie, “The machine as artist: An introduction,” in *Arts*, 2017, vol. 6, no. 2, p. 5.
- [10] I. Wallach, M. Dzamba, and A. Heifets, “AtomNet: a deep convolutional neural network for bioactivity prediction in structure-based drug discovery,” *arXiv Prepr. arXiv1510.02855*, 2015.
- [11] A. Zhavoronkov *et al.*, “Deep learning enables rapid identification of potent DDR1 kinase inhibitors,” *Nat. Biotechnol.*, vol. 37, no. 9, pp. 1038–1040, 2019.
- [12] X. Feng *et al.*, “The Deep Learning–Based Recommender System ‘Pubmender’ for Choosing a Biomedical Publication Venue: Development and Validation Study,” *J. Med. Internet Res.*, vol. 21, no. 5, p. e12957, 2019.
- [13] E. Choi, A. Schuetz, W. F. Stewart, and J. Sun, “Using recurrent neural network models for early detection of heart failure onset,” *J. Am. Med. Informatics Assoc.*, vol. 24, no. 2, pp. 361–370, 2017.
- [14] U. S. ARMY RESEARCH LABORATORY, “Army researchers develop new algorithms to train robots,” *EurekAlert*. https://www.eurekalert.org/pub_releases/2018-02/uarl-ard020218.php (accessed Jun. 28, 2020).
- [15] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning. nature 521 (7553), 436-444,” *Google Sch. Google Sch. Cross Ref Cross Ref*, 2015.
- [16] V. Vanhoucke, A. Senior, and M. Z. Mao, “Improving the speed of neural networks on

- CPU,” 2011.
- [17] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
 - [18] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun, “Pedestrian detection with unsupervised multi-stage feature learning,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 3626–3633.
 - [19] R. Zhao, W. Luk, X. Niu, H. Shi, and H. Wang, “Hardware acceleration for machine learning,” in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 645–650.
 - [20] A. Bansal, C. Coats, E. Lissos, and B. Schreiber, “Sidebar: Scratchpad Based Communication Between CPUs and Accelerators,” *arXiv Prepr. arXiv1910.10794*, 2019.
 - [21] Y. Chen *et al.*, “Dadiannao: A machine-learning supercomputer,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 609–622.
 - [22] Z. Zhang, M. Lyons, M. Schuster, and S. Akamatsu, “Comparison between geometry-based and gabor-wavelets-based facial expression recognition using multi-layer perceptron,” in *Proceedings Third IEEE International Conference on Automatic face and gesture recognition*, 1998, pp. 454–459.
 - [23] R. Salakhutdinov and G. Hinton, “An efficient learning procedure for deep Boltzmann machines,” *Neural Comput.*, vol. 24, no. 8, pp. 1967–2006, 2012.
 - [24] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, “Deep learning with COTS HPC systems,” in *International conference on machine learning*, 2013, pp. 1337–1345.
 - [25] Y. Le Cun *et al.*, “Handwritten digit recognition: Applications of neural network chips and automatic learning,” *IEEE Commun. Mag.*, vol. 27, no. 11, pp. 41–46, 1989.
 - [26] M. Manavazhahan, “A study of activation functions for neural networks,” 2017.
 - [27] B. Ding, H. Qian, and J. Zhou, “Activation functions and their characteristics in deep neural networks,” in *2018 Chinese Control And Decision Conference (CCDC)*, 2018, pp. 1836–1841.
 - [28] C. Szegedy *et al.*, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.
 - [29] G. H. Yann LeCun, Yoshua Bengio, “Deep Learning,” 2015.
 - [30] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, vol. 61, pp. 85–117, 2015.
 - [31] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, 1943.

- [32] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychol. Rev.*, vol. 65, no. 6, p. 386, 1958.
- [33] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," Cornell Aeronautical Lab Inc Buffalo NY, 1961.
- [34] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for Boltzmann machines," *Cogn. Sci.*, vol. 9, no. 1, pp. 147–169, 1985.
- [35] K. Fukushima, "Neocognitron: A hierarchical neural network capable of visual pattern recognition," *Neural networks*, vol. 1, no. 2, pp. 119–130, 1988.
- [36] Y. LeCun *et al.*, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, 1990, pp. 396–404.
- [37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [38] J. N. H. Heemskerk, "Overview of neural hardware. neurocomputers for brain-style processing. design, implementation and application," *PhD Thesis, Unit Exp. Theor. Psychol. Leiden Univ.*, 1995.
- [39] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.
- [40] M. N. C. S. Steve Clayton, Editor, "A breakthrough in speech recognition with Deep-Neural-Network approach." <https://blogs.microsoft.com/ai/a-breakthrough-in-speech-recognition-with-deep-neural-network-approach/>.
- [41] M. Z. Alom *et al.*, "The history began from alexnet: A comprehensive survey on deep learning approaches," *arXiv Prepr. arXiv1803.01164*, 2018.
- [42] T. Chen *et al.*, "A high-throughput neural network accelerator," *IEEE Micro*, vol. 35, no. 3, pp. 24–32, 2015.
- [43] A. Khan, A. Sohail, U. Zahoor, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artif. Intell. Rev.*, vol. 53, no. 8, pp. 5455–5516, 2020.
- [44] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Adv. Neural Inf. Process. Syst.*, vol. 25, pp. 1097–1105, 2012.
- [45] E. Thrun, Sebastian, and Lorien Pratt, *Learning to learn*. Springer Science & Business Media, 2012.
- [46] P. Wang, E. Fan, and P. Wang, "Comparative analysis of image classification algorithms based on traditional machine learning and deep learning," *Pattern Recognit. Lett.*, vol. 141, pp. 61–67, 2021.
- [47] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

- [48] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv Prepr. arXiv1409.1556*, 2014.
- [49] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [50] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017, vol. 31, no. 1.
- [51] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [52] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.
- [53] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.
- [54] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.
- [55] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.
- [56] B. Zoph, V. Vasudevan, J. Shlens, and Q. V Le, “Learning transferable architectures for scalable image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.
- [57] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv Prepr. arXiv1810.04805*, 2018.
- [58] M. Shoenberger, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using model parallelism,” *arXiv Prepr. arXiv1909.08053*, 2019.
- [59] B. Yann, LeCun, Christopher J.C, “THE MNIST DATABASE of Handwritten digits.” <http://yann.lecun.com/exdb/mnist/> (accessed Mar. 02, 2021).
- [60] A. Krizhevsky, “The CIFAR-10 dataset.” <https://www.cs.toronto.edu/~kriz/cifar.html> (accessed Mar. 03, 2021).
- [61] IMAGENET, “ImageNet Dataset.” <https://image-net.org/index.php> (accessed Mar. 03, 2021).

- [62] Gluon, “Prepare the ImageNet dataset.” https://cv.gluon.ai/build/examples_datasets/imagenet.html (accessed Mar. 03, 2021).
- [63] S. Rajan, “How will channels (RGB) effect convolutional neural network?,” 2019. <https://www.researchgate.net/post/How-will-channels-RGB-effect-convolutional-neural-network/5c67b72d36d23566117da1a0/citation/download> (accessed Mar. 02, 2021).
- [64] M. M. Islam and J.-M. Kim, “Vision-based autonomous crack detection of concrete structures using a fully convolutional encoder–decoder network,” *Sensors*, vol. 19, no. 19, p. 4251, 2019.
- [65] P. Solai, “Convolutions and Backpropagations,” *medium.com*. <https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c> (accessed Jan. 03, 2021).
- [66] X. Jin, “Note on Backpropagation in Neural Networks,” 2019.
- [67] S. Sathasivam and W. A. T. W. Abdullah, “Logic learning in Hopfield networks,” *arXiv Prepr. arXiv0804.4075*, 2008.
- [68] A. Afshine, Amidi, Shervine, “Recurrent Neural Networks cheatsheet,” *stanford.edu*. <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks> (accessed Jan. 05, 2021).
- [69] P. Liu, X. Qiu, and X. Huang, “Recurrent neural network for text classification with multi-task learning,” *arXiv Prepr. arXiv1605.05101*, 2016.
- [70] M. Wand, J. Koutník, and J. Schmidhuber, “Lipreading with long short-term memory,” in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 6115–6119.
- [71] S. Fernández, A. Graves, and J. Schmidhuber, “An application of recurrent neural networks to discriminative keyword spotting,” in *International Conference on Artificial Neural Networks*, 2007, pp. 220–229.
- [72] H. Zen and H. Sak, “Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 4470–4474.
- [73] I. Sutskever, O. Vinyals, and Q. V Le, “Sequence to sequence learning with neural networks,” *arXiv Prepr. arXiv1409.3215*, 2014.
- [74] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” *IEEE Trans. neural networks Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, 2016.
- [75] and F. C. Gers, Felix A, Jürgen Schmidhuber, “Learning to forget: Continual prediction with LSTM,” *Inst. Eng. Technol.*, pp. 850–855, 1999.
- [76] K. Cho *et al.*, “Learning phrase representations using RNN encoder-decoder for statistical

- machine translation,” *arXiv Prepr. arXiv1406.1078*, 2014.
- [77] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, “Light gated recurrent units for speech recognition,” *IEEE Trans. Emerg. Top. Comput. Intell.*, vol. 2, no. 2, pp. 92–102, 2018.
 - [78] Y. Su and C.-C. J. Kuo, “On extended long short-term memory and dependent bidirectional recurrent neural network,” *Neurocomputing*, vol. 356, pp. 151–161, 2019.
 - [79] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv Prepr. arXiv1412.3555*, 2014.
 - [80] S. Harris and D. Harris, *Digital design and computer architecture: arm edition*. Morgan Kaufmann, 2015.
 - [81] K. Potdar, T. S. Pardawala, and C. D. Pai, “A comparative study of categorical variable encoding techniques for neural network classifiers,” *Int. J. Comput. Appl.*, vol. 175, no. 4, pp. 7–9, 2017.
 - [82] S. K. Knapp, “Accelerate FPGA macros with one-hot approach,” *Electron. Des.*, vol. 38, no. 17, pp. 71–78, 1990.
 - [83] Xilinx, “HDL Synthesis for FPGAs Design Guide”. section 3.13:, Encoding State Machines, Accelerate FPGA Macros with One-Hot Approach,” 1995.
 - [84] V. Teller, “Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition,” *Comput. Linguist.*, vol. 26, no. 4, pp. 638–641, 2000.
 - [85] J. R. Firth, “A synopsis of linguistic theory, 1930–1935; studies in linguistic analysis,” *London Philos. Soc.*, 1962.
 - [86] A. Neelakantan, J. Shankar, A. Passos, and A. McCallum, “Efficient non-parametric estimation of multiple embeddings per word in vector space,” *arXiv Prepr. arXiv1504.06654*, 2015.
 - [87] Y. Goldberg and O. Levy, “word2vec Explained: deriving Mikolov et al.’s negative-sampling word-embedding method,” *arXiv Prepr. arXiv1402.3722*, 2014.
 - [88] S. Liu *et al.*, “Cambricon: An instruction set architecture for neural networks,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 393–405.
 - [89] B. Fleischer *et al.*, “A scalable multi-TeraOPS deep learning processor core for AI training and inference,” in *2018 IEEE Symposium on VLSI Circuits*, 2018, pp. 35–36.
 - [90] A. Shawahna, S. M. Sait, and A. El-Maleh, “FPGA-based accelerators of deep learning networks for learning and classification: A review,” *IEEE Access*, vol. 7, pp. 7823–7859, 2018.
 - [91] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, “Border control: Sandboxing

- accelerators,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 470–481.
- [92] H. Esmailzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, “Neural network stream processing core (NnSP) for embedded systems,” in *2006 IEEE International Symposium on Circuits and Systems*, 2006, pp. 4-pp.
 - [93] T. L. et al Chen, Yunji, Huiying Lan, Zidong Du, Shaoli Liu, Jinhua Tao, Dong Han, “An Instruction Set Architecture for Machine Learning,” *ACM Trans. Comput. Syst.*, vol. 3, no. 36, pp. 1–35, 2019.
 - [94] N. Wu, T. Jiang, L. Zhang, F. Zhou, and F. Ge, “A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set,” *Electronics*, vol. 9, no. 6, p. 1005, 2020.
 - [95] D. Valencia, S. F. Fard, and A. Alimohammad, “An artificial neural network processor with a custom instruction set architecture for embedded applications,” *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 67, no. 12, pp. 5200–5210, 2020.
 - [96] B. Hanin and D. Rolnick, “How to start training: The effect of initialization and architecture,” *arXiv Prepr. arXiv1803.01719*, 2018.
 - [97] S. K. Kumar, “On weight initialization in deep neural networks,” *arXiv Prepr. arXiv1704.08863*, 2017.
 - [98] Paperswithcode.com, “Kaiming Initialization,” *paperswithcode.com*. <https://paperswithcode.com/method/he-initialization> (accessed Mar. 06, 2021).
 - [99] Random.ORG, “Gaussian Random Number Generator,” *Random.ORG*. <https://www.random.org/gaussian-distributions/> (accessed Jan. 04, 2021).
 - [100] A. Rosebrock, “Keras Conv2D and Convolutional Layers,” *Pyimagesearch*. <https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/>.
 - [101] T. Tambe *et al.*, “Adaptivfloat: A floating-point based data type for resilient deep learning inference,” *arXiv Prepr. arXiv1909.13271*, 2019.
 - [102] K. Ogawa, Eri and Ishizaki, Kazuaki and Inoue, Hiroshi and Venkataramani, Swagath and Choi, Jungwook and Wang, Wei and Srinivasan, Vijayalakshmi and Ohara, Moriyoshi and Gopalakrishnan, “A Compiler for Deep Neural Network Accelerators to Generate Optimized Code for a Wide Range of Data Parameters from a Hand-crafted Computation Kernel,” in *2019 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, 2019, pp. 1–3.

Vitae

Name : Ali Omar Ali

Nationality : Tanzanian

Date of Birth :11/20/1990

Email : engineerbably@gmail.com

Address : King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia, Tel. +966509340039

Permanent Address : Zanzibar, Tanzania, ZIP: 00255, P.O.Box 280 Zanzibar, Tel No +255776291900

Academic Background : Bachelor of Computer Engineering, University Malaysia Sabah (UMS), Malaysia, November 2015