The Network Times
Handbook Series – Part X

Deep Learning for Network Engineers

Understanding Traffic Patterns and Network Requirements in the AI Data Center

Deep Learning uses Deep Neural Network (DNN) models to predict accurate outputs based on input data. To achieve reliable predictions, test datasets are run through the DNN over multiple iterations, during which model parameters are adjusted to improve performance. In large-scale training clusters, this process generates massive "elephant flows" between workers, requiring a high-speed, lossless transport network. Communication between workers may leverage intra-server high-speed NVLinks or an Ethernet-based backend network for inter-node traffic. This book not only explains Deep Learning processes but also explores how to design and build the high-performance backend networks needed to support them.

Toni Pasanen, CCIE#28158

The Network Times

Handbook Series – Part X

Deep Learning for Network Engineers Understanding Traffic Patterns and Network Requirements in the AI Data Center

Toni Pasanen, CCIE#28158

First Edition - 18 May 2025

Special thanks to Magnus Ekman for publishing the incredible book:

Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow, (17 Aug. 2021).

It was my primary reference when writing the Deep Learning chapters, which is why it's listed as the first source in Chapters 1–7.

This book is dedicated to my tiny black dog, who took me for long walks away from the computer—four to five times every day. And who graciously agreed to let me use her pretty face on the cover of my book once again, making this book not just mine, but ours.

ABOUT THE AUTHOR

Toni Pasanen, CCIE No. 28158 (Routing & Switching), is a Principal Network Architect and Distinguished Engineer at Fsas Technologies. He specializes in data center networking, including both traditional switched networks and modern BGP EVPN/VXLAN Clos fabrics, as well as AI data centers. In addition to on-premises data center networking, Toni is also an expert in cloud networking, particularly in Microsoft Azure and Amazon Web Services (AWS).

Although his current focus is on AI data center solutions, Toni has designed and implemented a wide range of networks across other domains, including MPLS-based WANs and data centers (e.g., MPLS-VPLS). He has also taught network technologies, such as Routing, Switching, MPLS, and QoS, while working for an official Cisco Learning Partner.

Since 2019, Toni has authored ten books on network technologies:

Virtual Extensible LAN – VXLAN:

A Practical Guide to VXLAN Solution,

August 2019

LISP Control-Plane in Campus Fabric:

A Practical Guide to Understand the Operation of Campus Fabric,

February 2020

VXLAN Fabric with BGP EVPN Control-Plane:

Design Considerations,

September 2020

Object-Based Approach to Cisco ACI:

The Logic Behind the Application Centric Infrastructure,

December 2020.

Cisco SD-WAN:

A Practical Guide to Understand the Basics of Cisco Viptela Based SD-WAN Solution, May 2021.

Network Virtualization:

LISP, OMP, and BGP EVPN Operation and Interaction,

August 2021.

AWS Networking Fundamentals:

A Practical Guide to Understand How to Build a Virtual Datacenter into the

AWS Cloud,

January 2021.

Azure Networking Fundamentals:

A Practical Guide to Understand How to Build a Virtual Datacenter into the Azure Cloud,

August 2022.

Azure Networking Handbook:

A Comprehensive Guide to Help You Step into the World of Azure Networking, August 2023.

Deep Learning for Network Engineer:

Understanding Traffic Patterns and Network Requirements in the AI Data Center,

May 2025.

ABOUT THIS BOOK

Several excellent books have been published over the past decade on Deep Learning (DL) and Datacenter Networking. However, I have not found a book that covers these topics together—as an integrated deep learning training system—while also highlighting the architecture of the datacenter network, especially the backend network, and the demands it must meet. This book aims to bridge that gap by offering insights into how Deep Learning workloads interact with and influence datacenter network design.

SO, WHAT IS DEEP LEARNING?

Deep Learning is a subfield of Machine Learning (ML), which itself is a part of the broader concept of Artificial Intelligence (AI). Unlike traditional software systems where machines follow explicitly programmed instructions, Deep Learning enables machines to learn from data without manual rule-setting.

At its core, Deep Learning is about training artificial neural networks. These networks are mathematical models composed of layers of artificial neurons. Different types of networks suit different tasks—Convolutional Neural Networks (CNNs) for image recognition, and Large Language Models (LLMs) for natural language processing, to name a few.

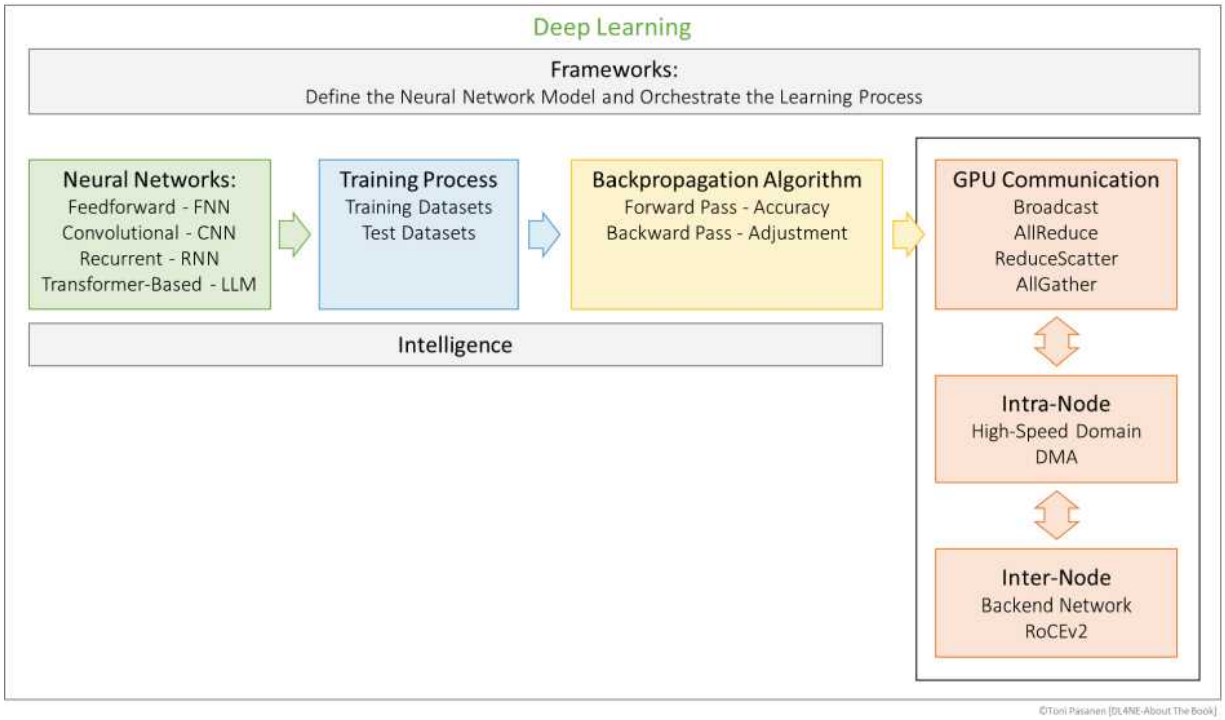
Training a neural network involves feeding it labeled data and adjusting its internal parameters through a process called backpropagation. During the forward pass, the model makes a prediction based on its current parameters. This prediction is then compared to the correct label to calculate an error. In the backward pass, the model uses this error to update its parameters, gradually improving its predictions. Repeating this process over many iterations allows the model to learn from the data and make increasingly accurate predictions.

WHY SHOULD NETWORK ENGINEERS CARE?

Modern Deep Learning models can be extremely large, often exceeding the memory capacity of a single GPU or CPU. In these cases, training must be distributed across multiple processors. This introduces the need for highspeed communication between GPUs—both within a single server (intranode) and across multiple servers (inter-node).

Intra-node GPU communication typically relies on high-speed interconnects like NVLink, with Direct Memory Access (DMA) operations enabling efficient data transfers between GPUs. Inter-node communication, however, depends on the backend network, either InfiniBand or Ethernet-based. Synchronization of model parameters across GPUs places strict requirements on the network: high throughput, ultralow latency, and zero packet loss. Achieving this in an Ethernet fabric is challenging but possible.

This is where datacenter networking meets Deep Learning. Understanding how GPUs communicate and what the network must deliver is essential for designing effective AI data center infrastructures.


Deep Learning
Frameworks:
Define the Neural Network Model and Orchestrate the Learning Process
Neural Networks:
Feedforward - FNN
Convolutional - CNN
Recurrent - RNN
Transformer-Based - LLM
Training Process
Training Datasets
Test Datasets
Backpropagation Algorithm
Forward Pass - Accuracy
Backward Pass - Adjustment
GPU Communication
Broadcast
AllReduce
ReduceScatter
AllGather
Intra-Node
High-Speed Domain
DMA
Inter-Node
Backend Network
RoCEv2
Intelligence
©Toni Pasanen (DL4NE-About The Book)

WHAT THIS BOOK IS—AND ISN'T

This book provides a theoretical and conceptual overview. It is not a configuration or implementation guide, although some configuration examples are included to support key concepts. Since the focus is on the Deep Learning process, not on interacting with or managing the model, there are no chapters covering frontend or management networks. The storage network is also outside the scope. The focus is strictly on the backend network.

The goal is to help readers—especially network professionals—grasp the "big picture" of how Deep Learning impacts data center networking.

ONE FINAL NOTE

In all my previous books, I've used font size 10 and single line spacing. For this book, I've increased the font size to 11 and the line spacing to 1.15. This wasn't to add more pages but to make the reading experience more comfortable. I've also tried to ensure that figures and their explanations appear on the same page, which occasionally results in some white space.

I hope you find this book helpful and engaging as you explore the fascinating intersection of Deep Learning and Datacenter Networking.

HOW THIS BOOK IS ORGANIZED

PART I – CHAPTERS 1-8: DEEP LEARNING AND DEEP NEURAL NETWORKS

This part of the book lays the theoretical foundation for understanding how modern AI models are built and trained. It introduces the structure and purpose of artificial neurons and gradually builds up to complete deep learning architectures and parallel training methods.

Artificial Neurons and Feedforward Networks (Chapters 1 - 3)

The journey begins with the artificial neuron, also known as a perceptron, which is the smallest functional unit of a neural network. It operates in two key steps: performing a matrix multiplication between inputs and weights, followed by applying a non-linear activation function to provide an output.

By connecting many neurons across layers, we form a Feedforward Neural Network (FNN). FNNs are ideal for basic classification and regression tasks and provide the stepping stone to more advanced architectures.

Specialized Architectures: CNNs, RNNs, and Transformers (Chapters 3 - 9)

After covering FNNs, this part dives into models designed for specific data types:

- Convolutional Neural Networks (CNNs): Optimized for spatial data like images, CNNs use filters to extract local features such as edges, textures, and shapes, while keeping the model size efficient through weight sharing.
- Recurrent Neural Networks (RNNs): Designed for sequential data like text and time series, RNNs maintain a hidden state that captures previous input history. This allows them to model temporal dependencies and context across sequences.
- Transformer-based Large Language Models (LLMs): Unlike RNNs, Transformers use self-attention mechanisms to weigh relationships between all tokens in a sequence simultaneously. This architecture underpins state-of-the-art language models and enables scaling to billions of parameters.

Parallel Training and Scaling Deep Learning (Chapter 8)

As models and datasets grow, training them on a single GPU becomes impractical. This section explores the three major forms of distributed training:

- Data Parallelism: Each GPU holds a replica of the model but processes different mini-batches of input data. Gradients are synchronized at the end of each iteration to keep weights aligned.
- Pipeline Parallelism: The model is split across multiple GPUs, with each GPU handling one stage of the forward and

backward pass. Micro-batches are used to keep the pipeline full and maximize utilization.

- Tensor (Model) Parallelism: Very large model layers are broken into smaller slices, and each GPU computes part of the matrix operations. This approach enables the training of ultra-large models that don't fit into a single GPU's memory.

PART II – CHAPTERS 9 – 14: AI DATA CENTER NETWORKING

This part of the book focuses on the network technologies that enable distributed training at scale in modern AI data centers. It begins with an overview of GPU-to-GPU memory transfer mechanisms over Ethernet and then moves on to congestion control, load balancing strategies, network topologies, and GPU communication collectives.

RoCEv2 and GPU-to-GPU Transfers (Chapter 9)

The section starts by explaining how Direct Memory Access (DMA) is used to copy data between GPUs across Ethernet using RoCEv2 (RDMA over Converged Ethernet version 2). This method allows GPUs located in different servers to exchange large volumes of data without CPU involvement.

DCQCN: Data Center Quantized Congestion Notification (Chapters 10 - 11)

RoCEv2's performance depends on a lossless transport layer, which makes congestion management essential. To address this, DCQCN provides an advanced congestion control mechanism. It dynamically adjusts traffic flow based on real-time feedback from the network to minimize latency and packet loss during GPU-to-GPU communication.

- Explicit Congestion Notification (ECN): Network switches mark packets instead of dropping them when congestion builds. These marks trigger rate adjustments at the sender to prevent overload.
- Priority-based Flow Control (PFC): PFC ensures that traffic classes like RoCEv2 can pause independently, preventing buffer overflows without stalling the entire link.

Load Balancing Techniques in AI Traffic (Chapter 12)

In addition to congestion control, effective load distribution is critical for sustaining GPU throughput during collective communication. This section introduces several techniques used in modern data center fabrics:

- Flow-based Load Balancing: Assigns entire flows or flowlets to paths based on real-time link usage or hash-based distribution, improving path diversity and utilization.
- Flowlet Switching: Divides a flow into smaller time-separated bursts ("flowlets") that can be load-balanced independently without reordering issues.
- Packet Spraying: Distributes packets belonging to the same flow across multiple available paths, helping to avoid link-level bottlenecks.

AI Data Center Network Topologies (Chapter 13)

Next, the section discusses design choices in the East-West fabric—the internal network connecting GPU servers. It

introduces topologies such as:

- Top-of-Rack (ToR): Traditional rack-level switching used to connect servers within a rack.
- Rail and Rail-Optimized Designs: High-throughput topologies tailored for parallel GPU clusters. These layouts improve resiliency and throughput, especially during bursty communication phases in training jobs.

GPU-to-GPU Communication (Chapter 14)

The part concludes with a practical look at collective communication patterns used to synchronize GPUs across the network. These collectives are essential for distributed training workloads:

- AllReduce: Each GPU contributes and receives a complete, aggregated copy of the data. Internally, this is implemented in two phases:

1. ReduceScatter: GPUs exchange partial results and compute a portion of the final sum.
2. AllGather: Each GPU shares its computed segment so that every GPU receives the complete aggregated result.

- Broadcast: A single GPU (often rank 0) sends data—such as communication identifiers or job-level metadata—to all other GPUs at the start of a training job.

TARGET AUDIENCE

I wrote this book for professionals working in the data center networking domain—whether in architectural, design, or specialist roles. It is especially intended for those who are already involved in, or are preparing to work with, the unique demands of AI-driven data centers. As AI workloads reshape infrastructure requirements, this book aims to provide the technical grounding needed to understand both the deep learning models and the networking systems that support them.

DISCLAIMERS

The content of this book is based solely on the author's personal experience, research, and testing. It has not been reviewed, validated, or endorsed by Cisco, NVIDIA, or any other organization or individual. This book is not intended to serve as a design or implementation guide. Readers are encouraged to perform their own validation and testing before applying any of the concepts or techniques in a production environment.

TABLE OF CONTENTS

About the Author iv

About This Book vi

So, what is Deep Learning? vi

Why should network engineers care? vii

What this book is—and isn't viii

One final note ix

How this book is organized x

Part I – Chapters 1-8: Deep Learning and Deep Neural Networks x

Part II – Chapters 9 – 14: AI Data Center Networking xii

Target Audience xv

Disclaimers xv

Table of Contents xvi

Chapter 1: Artificial Neuron 1

Introduction 1

Artificial Neuron 3

Weighted Sum and Activation Function 4

Bias term 8

ReLU Activation Function 10

Network Impact 11

Summary 12

References 13

Chapter 2: Backpropagation Algorithm 15

Introduction 15

Forward Pass 16

Learning Rate 19

Backward Pass 20

Partial Derivative for Error Function – Output Error 21

Partial Derivative for the Activation Function 23

Error Term for Neurons 23

Gradient Calculation 24

Weight Adjustment 26

The Second Iteration - Forward Pass 26

Network Impact 28

References 30

Chapter 3: Multi-Class Classification 31

Introduction 31

MNIST Dataset 31

Forward Pass 33

Model Probability 33

Cross-Entropy Loss 35

Backward Pass 36

Gradient Computing 36

Weight Adjustment Values 37

Weight Update 38

References 41

Chapter 4: Convolutional Neural Networks 43

Introduction 43

Convolution Layer 44

Convolution Process 44

MaxPooling 48

The First Convolution Layer: *Convolution* 48

The First Convolution Layer: *MaxPooling* 51

The Second Convolution Layer 52

Fully Connected Layers 53

Backpropagation Process 53

References 55

Chapter 5: Recurrent Neural Networks 57

Introduction 57

Text Datasets 58

Training Recurrent Neural Networks 59

Weight Matrices in RNNs 59

Weighted Sum Calculation in the Hidden Layer 59

Output Layer Operations 60

Comparison with Feed-Forward Neural Networks (FNNs) 61

Moving to the Second Time Step 61

Backward Pass in Recurrent Neural Networks 61

Challenges of a RNN Modell 62

Saturated Neurons 64

References 68

Chapter 6: Long Short-Term Memory 69

Introduction 69

LTSM cell operation 71

Forget Gate 71

Input Gate 73

Output Gate 74

LTSM-based Recurrent Neural Network 75

Recap of the Operation of an LSTM Cell 75

An Overview of an LSTM-Based RNN 76

Conclusion 79

References 80

Chapter 7: Large Language Model (LLM) 81

Introduction 81

Tokenizer and Word Embedding Matrix 81

Word Embedding 83

Positional Embeddings 87

Calculating the Final Word Embedding 88

Transformer Architecture 90

Introduction 90

Query, Key and Value Vectors 92

Attention Layer 94

Add & Normalization 96

Feed Forward Neural Network 97

Next Word Probability Computation – SoftMax Function 98

Conclusion 102

References 103

Chapter 8: Parallelism Strategies in Deep Learning 105

Introduction 105

Data Parallelism 106

Model Parallelism with Pipeline Parallelism 108

1st. Time Step — Active GPUs: 25% - Idle GPUs: 1a, 0b, 1b 109

2nd. Time Step — Active GPUs: 50% - Idle GPUs: 0b, 1b 110

3rd. Time Step— Active GPUs: 75% - Idle GPUs: 1b 111

4th. Time Step— Active GPUs: 100% - Idle GPUs: none 112

5th. Time Step — Active GPUs: 75% - Idle GPUs: 0a 113

6th. Time Step — Three Active GPUs, One Idle GPU: Overall GPU Utilization 75% 114

7th. Time Step — Four Active GPUs: Overall GPU Utilization 100% 115

8th. Time Step — Three Active GPUs: One Idle GPU: Overall GPU Utilization 75% 115

9th. Time Step — Two Active GPUs: Two Idle GPUs: Overall GPU Utilization 50% 116

10th. Time Step — One Active GPU: three Idle GPUs: Overall GPU Utilization 25% 116

Tensor Parallelism 117

Self-Attention Layer 119

Feedforward Layer 122

Backpropagation 123

References 127

Chapter 9: RDMA Basics 129

Introduction 129

An overview of RDMA Processes 130

Memory Allocation and Registration 131

Create Queue Pairs 132

RDMA Connection Initiation 134

Work Request Message 135

References 138

Chapter 10: Challenges in AI Fabric Design 139

Introduction 139

Egress Interface Congestions 140

Single Point of Failure 141

Head-of-Line Blocking 143

Hash-Polarization with ECMP 144

References 146

Chapter 11: Congestion Avoidance 147

GPU-to-GPU RDMA Write Without Congestion 148

Explicit Congestion Notification -ECN 149

DSCP-Based Priority Flow Control (PFC) 153

Step 1: Buffer Overflow on Rail Switch C (Egress to GPU-3, Host 3) 154

Step 2: xOFF Threshold Exceeded 155

Step 3: Generating a PFC Pause Frame (MAC Control Frame) 155

Step 4: Spine Switch 1 Pauses Transmission on Priority Queue 3 156

Step 5: Congestion on Spine Switch 1 Egress Queue to Rail Switch C 157

Step 6: xOFF Threshold Exceeded on Spine Switch 1 Ingress Interfaces 158

Step 7: Spine Switch 1 Sends PFC Pause Frames to Rail Switch A and B 158

Step 8: Rail Switches A and B Pause Queue 3 Traffic to Spine Switch 1 158

Steps 9a – 14: Downstream Resume and Congestion Recovery 159

LLDP with DCBX 161

Data Center Quantized Congestion Notification (DCQCN) 162

How DCQCN Combines ECN and PFC 162

Why ECN Must Precede xOFF 163

DCQCN Configuration 166

References 174

Chapter 12: Flow, Flowlet, and Packet-Based Load Balancing 175

Introduction 175

RDMA WRITE Operation 176

RDMA Write First 176

RDMA Write Middle 178

RDMA Write Last 179

Flow-Based Load Balancing with Layer 3 ECMP 180

Flowlet-Based Load Balancing with Adaptive Routing 181

Packet-Based Load Balancing with Packet Spraying 183

RDMA Write Only 184

Configuring Per-Packet Load Balancing on Cisco Nexus Switches 185

References 186

Chapter 13: Backend Network Topologies 187

Introduction 187

Shared NIC 187

NIC per GPU 190

Design Scenarios 193

Single Rail Switch Design with Dedicated, Single-Port NICs per GPU 193

Dual-Rail Switch Topology with Dedicated, Dual-Port NICs per GPU 195

Cross-Rail Communication over NVLink in Rail-Only Topologies 199

Rail Desings in GPU Fabric 205

AI Fabric Architecture Conclusion 209

Hash Polarization 212

References 214

Chapter 14: GPU Cluster Communication Model 215

Distributing NCCL Unique Id for GPUs in a Training Cluster 217

Opening TCP Socket to Master Node 217

Distributing the NCCL Unique ID Over Established TCP Sockets 219

NCCL Broadcast Collective and Model Parameter Synchronization 220

Gradient Synchronization Using AllReduce Collective 222

References 238

Back Cover Text 240

CHAPTER 1:
ARTIFICIAL NEURON

INTRODUCTION

Before diving into the somewhat complex world of deep learning, let's first consider how humans learn new skills through repetition, feedback, and guidance.

Judo, as a martial art, serves as a good example. I trained in judo for over 20 years. During that time, I learned which throwing techniques to use to take down an opponent efficiently by leveraging their movement, energy, and reactions. But how did I learn that?

Through a supervised training process.

Our coach first taught us the throwing techniques and explained the situations in which they work best. We then practiced them, starting with static drills where the opponent stood still. Once the basic movement was learned, we moved on to pre-arranged sequences that introduced movement and timing.

Mastering these techniques required thousands of repetitions. Even then, perfect performance was not guaranteed, due to variables such as the opponent's movement, strength, body length, technical level, and so on.

After internalizing several throwing techniques, I reached a point where I could apply them in situations I hadn't

encountered before. That's when I was ready to compete, testing my skill in real matches, under the control of a referee who judged and scored the techniques.

How does this relate to Deep Learning?

Deep Learning (DL) is a driving force behind many of today's breakthroughs in artificial intelligence. It relies on Deep Neural Networks (DNNs)—systems made up of interconnected artificial neurons that learn

to recognize patterns in data. Deep Learning is a subfield of Machine Learning (ML), which enables computers to learn from data and make predictions or decisions without being explicitly programmed for every scenario. In turn, Machine Learning is part of the broader field of Artificial Intelligence (AI), which focuses on creating systems that can perform tasks that normally require human intelligence.

Training a neural network follows similar principles to learning a judo throwing technique.

In judo, my coach first taught me the correct technique: where to place my hands, how to shift my weight, and when to execute the throw. This is like supervised learning, where a neural network is trained with labeled data, input examples paired with the correct output, just like the coach provides correct demonstrations and feedback.

After I had practiced the technique in a static, controlled environment, we introduced more complexity by adding movement. I had to react to the opponent's motion and timing.

While the feedback was still there, the learning was now more intuitive and situational, similar to how a model can later benefit from unlabeled data or semi-supervised methods to generalize better.

Eventually, I entered competitions. There, I faced unpredictable situations, opponents with different styles, and no second chances. This is the equivalent of putting a trained neural network into production, where it must perform accurately on real-world data it hasn't seen before, without further coaching.

Like judo training, training a neural network takes time and repetition, often tens of thousands of iterations. After each iteration, the model's output is compared to the expected result, and its internal parameters are adjusted to reduce error. The process depends on several factors: dataset size, network architecture, hardware, and parallelization strategies. While training may take days or even months, the outcome is a system that can

respond quickly and reliably, just like a skilled judoka reacting instinctively in a match.

The duration of the training process depends on several factors, such as dataset size, network architecture, hardware, and selected parallelization strategies (if need). Training a neural network requires multiple iterations, sometimes even tens of thousands, where, at the end of each iteration, the model's output is compared to the actual value. If the difference between these two values is not small enough, the network is adjusted to improve performance. The entire process may take

months, but the result is a system that responds accurately and quickly, providing an excellent user experience.

This chapter begins by discussing the artificial neuron, and its functionality. We then move on to the Feedforward Neural Network (FFNN) model, first explaining its layered structure and how input data flows through it in a process called the Forward Pass (FP). Next, we examine how the FFNN is adjusted during the Backward Pass (BP), which fine-tunes the model by minimizing errors. The combination of FP and BP is known as the Backpropagation Algorithm.

ARTIFICIAL NEURON

An artificial neuron, also known as a perceptron, is a fundamental building block of any neural network. It functions as a computational unit that processes input data in two phases. First, it collects and processes all inputs (matrix multiplication), and then applies an activation function. Figure 1-1 illustrates the basic process without the complex mathematical functions (which I will explain later for those interested in studying them). On the left-hand side, we have a bias term and two input values, x1 and x2. The bias and inputs are connected to the perceptron through adjustable weight parameters: w0, w1, and w2, respectively. During the initial training phase, weight values are randomly generated.

Weighted Sum and Activation Function

As the first step, the neuron calculates the weighted sum of inputs x1 and x2 and adds the bias. A weighted sum simply means that each input is multiplied by its corresponding weight parameter, the results are summed, and the bias is added to the total. The bias value is set to one, so its contribution is always equal to the value of its weight parameter. I will explain the purpose of the bias term later in this chapter. The result of the weighted sum is denoted as z, which serves as a pre-activation

value. This value is then passed through a non-linear activation function, which produces the actual output of the neuron, $yy^$ (y-hat). Before explaining what non-linearity means in the context of activation functions and why it is used, consider the following: The input values fed into a neuron can be any number between negative infinity (-∞) and positive infinity (+∞). Additionally, there may be thousands of input values. As a result, the weighted sum can become a very large positive or negative value.

Now, think about neural networks with thousands of neurons. In Feedforward Neural Networks (FFNNs), neurons are structured into layers: an input layer, one or more hidden layers, and an output layer. If input values were only processed through the weighted sum computation and passed to the next layer, the neuron outputs would grow linearly with each layer. Even if we applied a linear activation function, the same issue would persist—the output would continuously increase. With a vast number of neurons and large input values, this uncontrolled growth could lead to excessive computational demands, slowing down the training process. Non-linear activation functions help keep output values within a manageable range. For example, an S-shaped Sigmoid activation function squeezes the neuron's output to a range between 0 and 1, even for very large input values.

Let's go back to Figure 1-1, where we first multiply the input values by their respective weight parameters, sum them, and then add the bias. Since the bias value is 1, it is reasonable to

represent it using only its associated weight parameter in the formula. If we plot the result z on the horizontal

axis of a two-dimensional chart and draw a vertical line upwards, we obtain the neuron's output value y at the point where the line intersects the S-curve. Simple as that. Naturally, there is a mathematical definition and equation for this process, which is depicted in Figure 1-2.

Before moving on, there is one more thing to note. In the figure below, each weight has an associated adjustment knob. These knobs are simply a visual representation to indicate that weight values are adjustable parameters, which will be tuned by the backpropagation algorithm in case the model output is not close enough to expected result. The backpropagation process is covered in detail in a dedicated chapter.

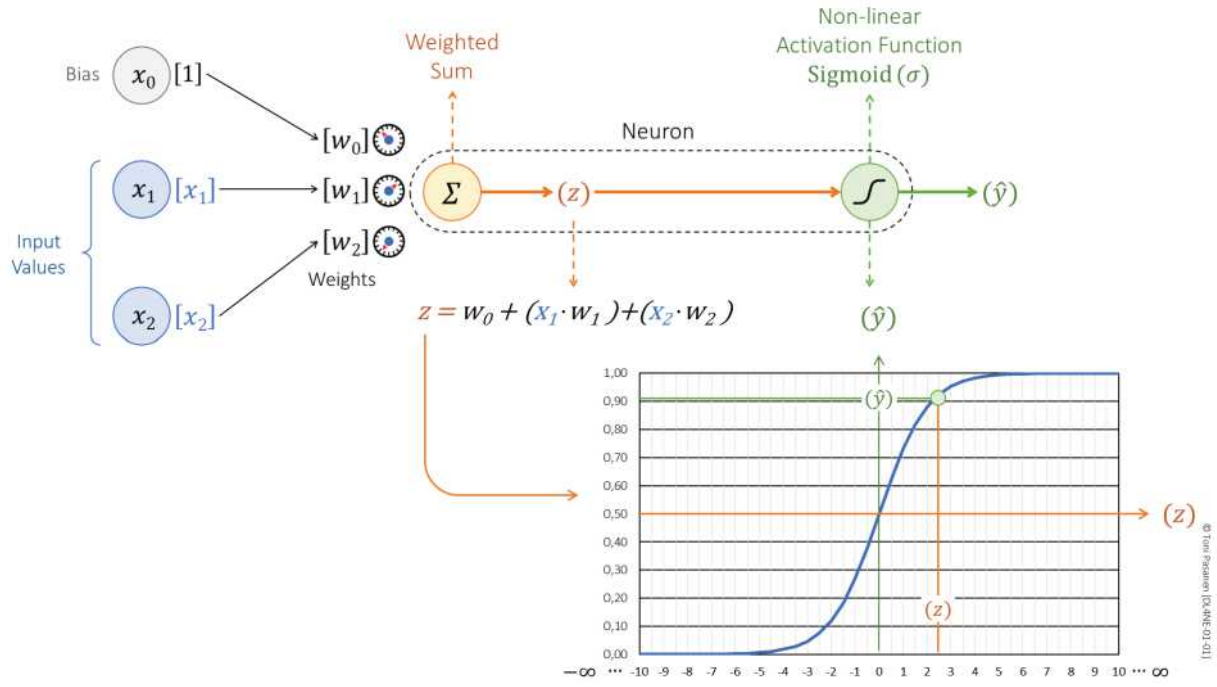


Figure 1-1: *An Architecture of an Artificial Neuron.*

Figure 1-2 shows the mathematical equations for calculating the weighted sum and the Sigmoid function. The Greek letter used in the weighted sum equation is Σ (uppercase Sigma). The lowercase i is set to 1 beneath the Sigma symbol, indicating that the weighted sum calculation starts from the first pair of elements: input x1 and its corresponding weight w1. The notation n=2 specifies the total number of paired elements included in the weighted sum calculation. In our example, both input values and their respective weights are included.

After computing the weighted sum, we add the bias term. The result, z, is then passed through the Sigmoid function, producing the output $yy\hat{}$. The Sigmoid function is commonly represented by the Greek letter σ (lowercase sigma).

The lower equation in Figure 1-2 shows how the Sigmoid function is computed. To obtain the denominator for the fraction, Euler's number ($e \approx 2.71828$) is raised to the power of -z and then summed with 1. The final output is simply the reciprocal of this sum.

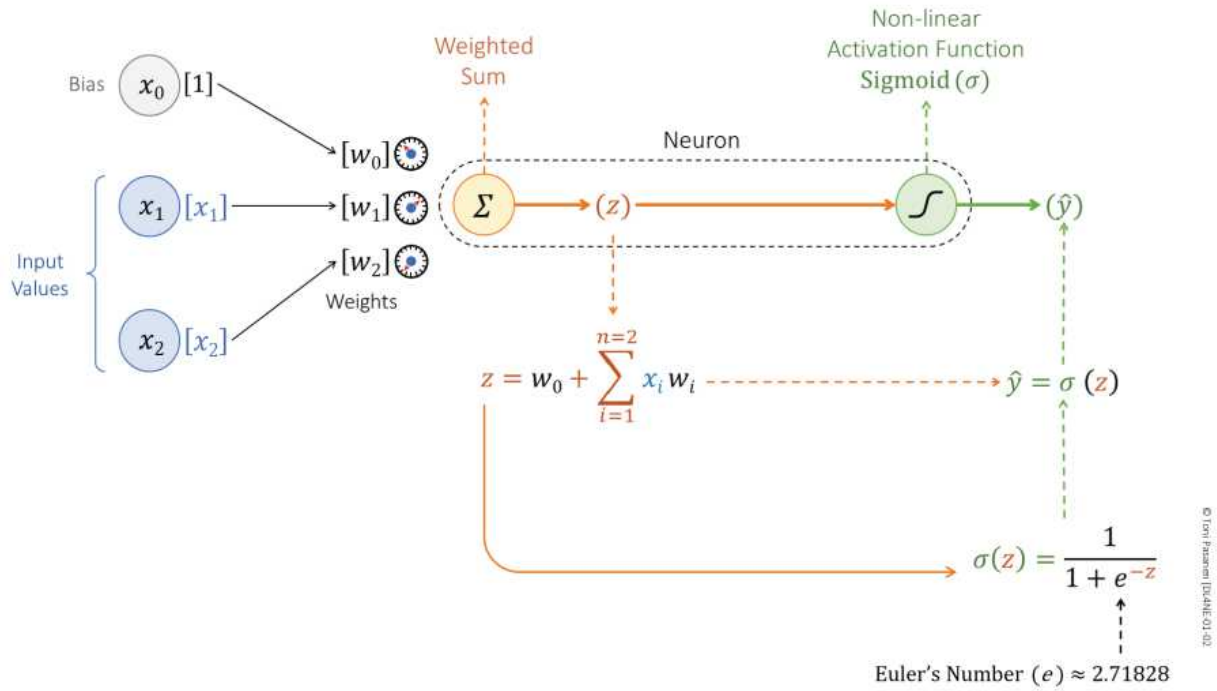


Figure 1-2: *The Math Behind an Artificial Neuron.*

The formulas can be expressed in an even simpler manner using dot products, which are commonly used in linear algebra. Dot products frequently appear in research papers and deep learning literature.

In Figure 1-3, both input values and weights are arranged as column vectors. The notation for the input vector uses an uppercase X, while the weight vector is denoted by an uppercase W. Although these are technically vectors, it is not a major issue to illustrate them as a simple matrix for demonstration purposes. Generally speaking, a matrix has more than one row and column, as you will learn later.

The dot product performs a straightforward matrix multiplication, as shown in the figure. This greatly simplifies the computation.

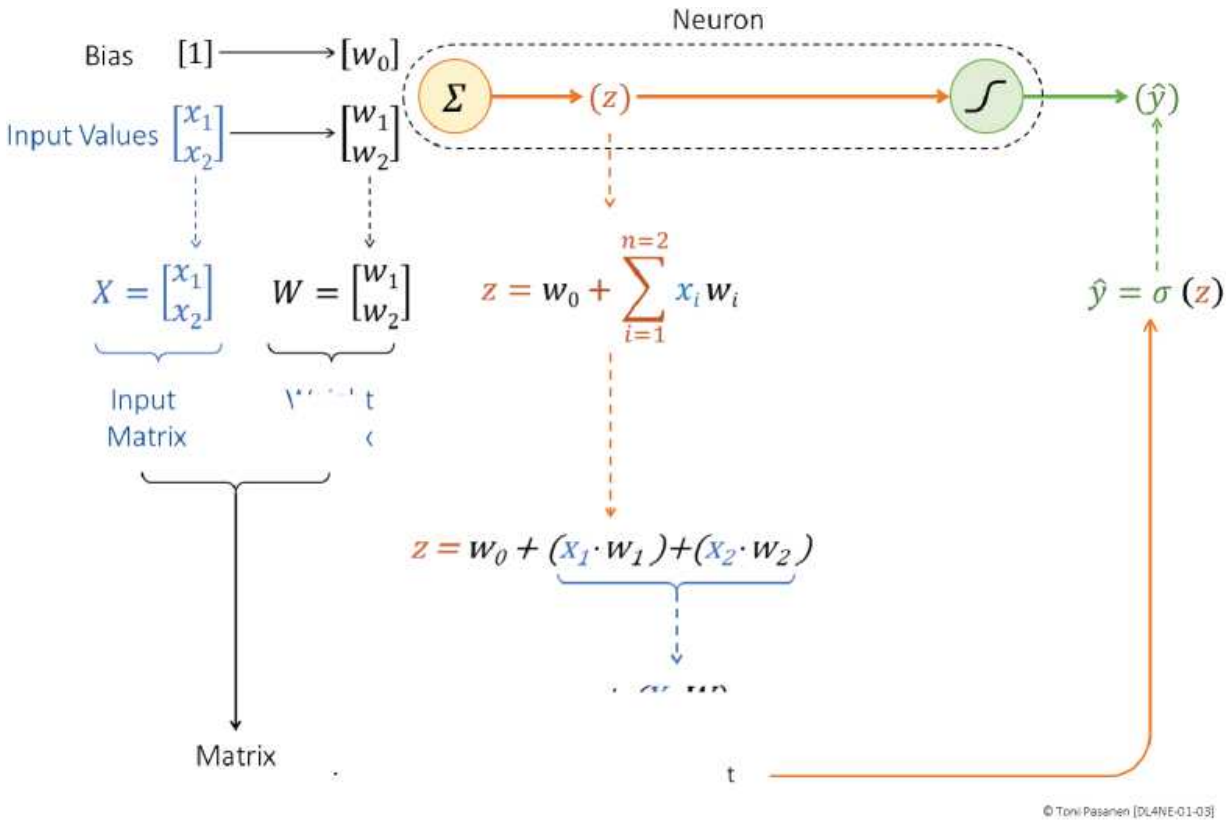


Figure 1-3: *Matrix Multiplication with Dot Product.*

Bias term

Figures 1-4 and 1-5 illustrate how changes in the bias weight parameter affect the weighted sum and shift z horizontally. This, in turn, changes the output of the Sigmoid function and the neuron's final output.

In Figure 1-4, the initial weight values for the bias, input x1, and input x2 are +0.3, +0.5, and -0.3, respectively. The calculated weighted sum is z=4.8. Applying the Sigmoid function to output z=4.8, we obtain an output value of 0.992. Figure 1-4 visualizes this process: z=4.8 is positioned on the horizontal axis, and the intersection with the S-curve results in an output of 0.992.

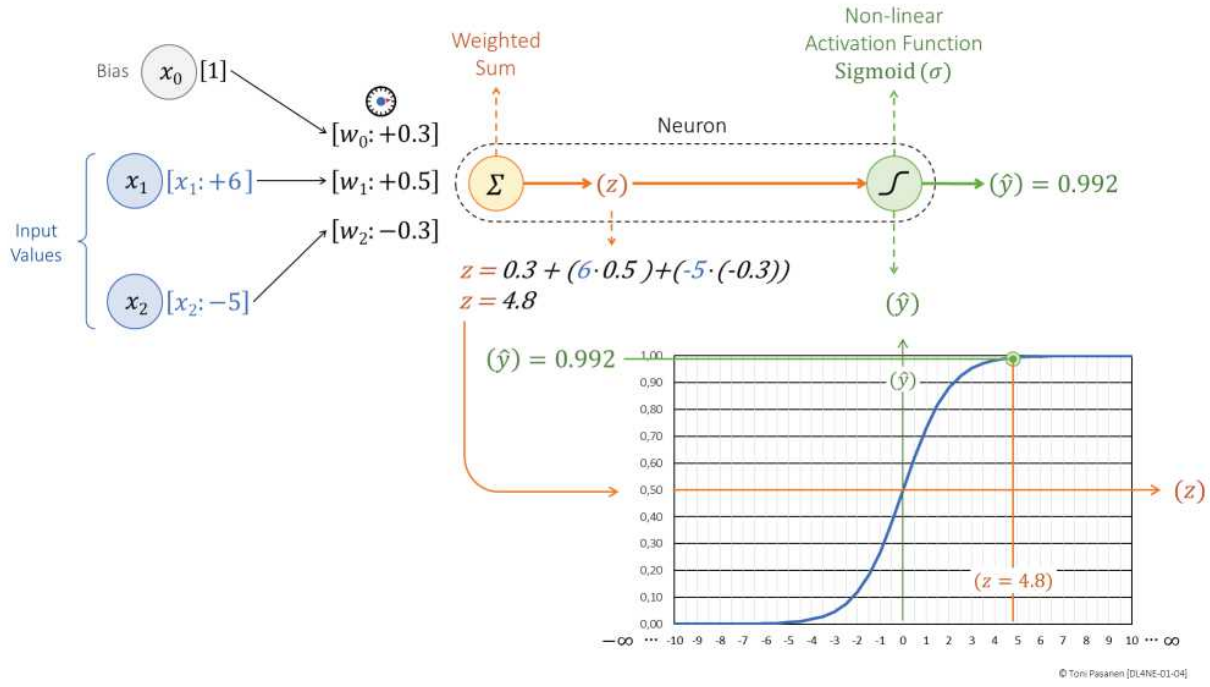


Figure 1-4: *Construct of an Artificial Neuron.*

Now, we adjust the weight wo associated with the bias from +0.3 down to -4.0. As a result, the weighted sum decreases from 4.84 to 0.50, shifting z 4.3 steps to the left on the horizontal axis. Applying the Sigmoid function to z, the neuron's output decreases from 0.992 to 0.622.

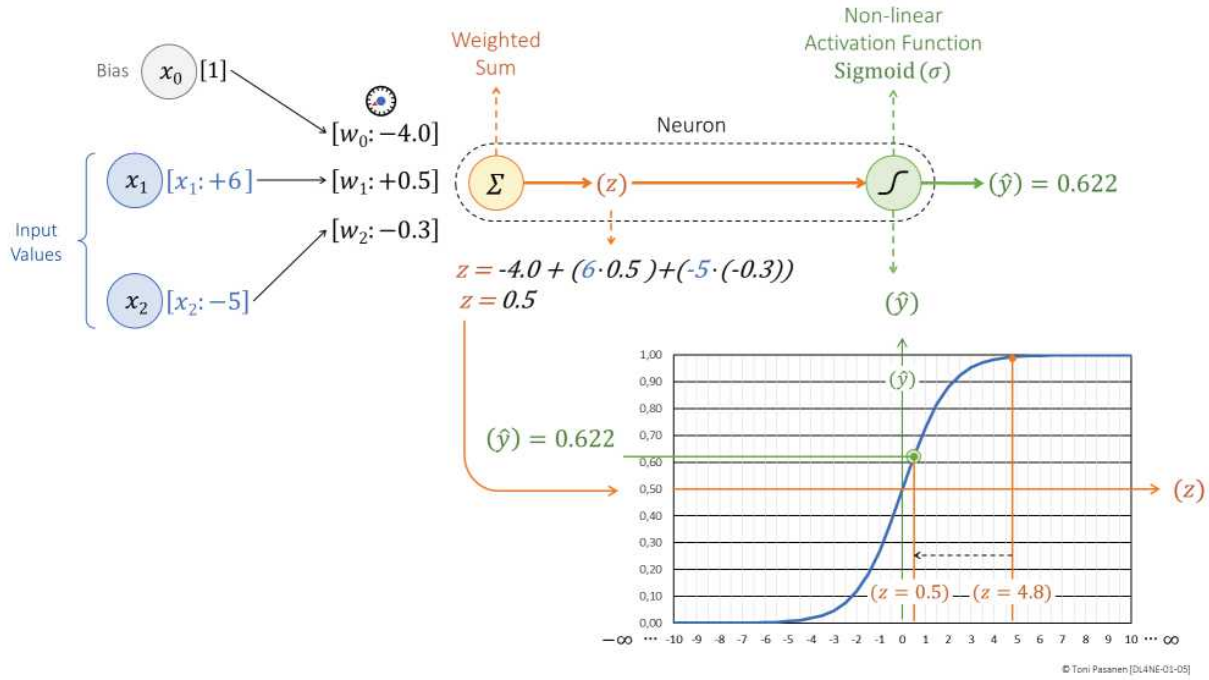


Figure 1-5: *Construct of an Artificial Neuron.*

In the example calculation above, imagine that input values x1 and x2 are zero. Without a bias term, the activation value will be zero, regardless of how large the weight parameters are. Therefore, the bias term also allows the neuron to produce non-zero outputs, even when all input values are

zero.

ReLU Activation Function

A lighter alternative to the Sigmoid activation function is ReLU (Rectified Linear Unit). The ReLU activation function is non-linear for values less than or equal to zero and linear for values greater than zero. This means that if the weighted sum $z \leq 0$, the output is zero. If $z > 0$, the output is equal to z.

From a computational perspective, ReLU requires fewer CPU cycles than the Sigmoid function. Figure 1-6 illustrates how z = 4.8 is processed by ReLU, resulting in an output value of $yy^{\wedge}$ = 4.8. The figure also shows two common notations for ReLU. The first notation states:

- If z > 0, return z.
- If z ≤ 0, return 0.

The second notation, written as MAX(0,z), simply means selecting the greater value between 0 and z.

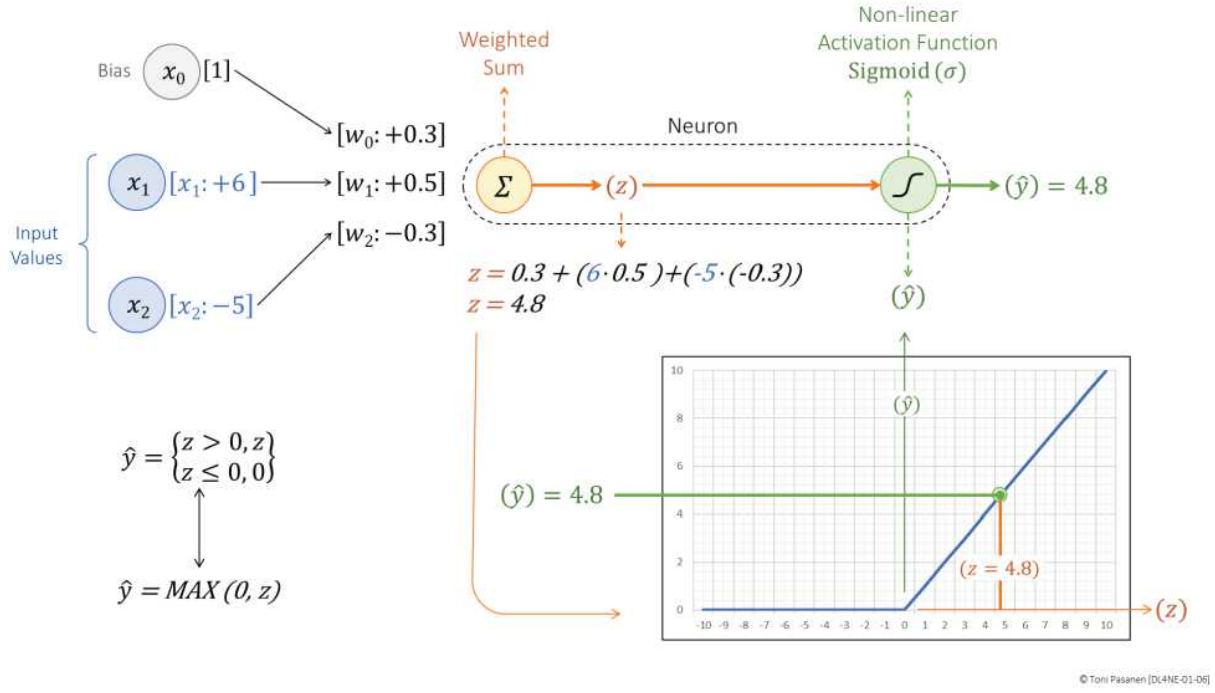


Figure 1-6: *Artificial Neuron with a ReLU Activation Function.*

NETWORK IMPACT

A single artificial neuron is the smallest unit of a neural network. The size of the neuron depends on its connections to input nodes. Every connection has an associated weight parameter, which is typically a 32-bit value. In our example, with 2 connections and bias, the size of the neuron is 3 x 32 bits = 96 bits.

Although we haven't defined the size of the input in this example, let's assume that each input (x) is an 8-bit value, giving us 2 x 8 bits = 16 bits for the input data. Thus, our single neuron "model" requires 96 bits for the weights plus 16 bits for the input data, totaling 112 bits of memory. This is small enough to not require parallelization. Besides, the weight parameters and input values, the result of weighted sum and the neuron output must be stored for processing.

However, if the memory requirement of the neural network model combined with the input data exceeds the memory capacity of a GPU, a parallelization strategy is needed. The data can be split across multiple GPUs within a single server, with synchronization happening over highspeed NVLink. If the job must be divided between multiple GPU servers, synchronization occurs over the backend network, which must provide lossless, high-speed packet forwarding.

Parallelization strategies will be discussed in the next chapter, which introduces a Feedforward Neural Network using the Backpropagation algorithm, and in later chapters dedicated to Parallelization (Chapter 8).

SUMMARY

Deep Learning leverages Neural Networks, which consist of artificial neurons. An artificial neuron mimics the structure and operation of a biological neuron. Input data is fed to the neuron through connections, each with its own weight parameter. The neuron uses these weights to calculate a weighted sum of the inputs, known as the pre-activation value. This result is then passed through an activation function, which provides the post-activation value, or the actual output of the neuron. The activation functions discussed in this chapter are the non-linear ReLU (Rectified Linear Unit) and logistic Sigmoid functions.

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Yann LeCun, Corina Cortes, Christoper J.C. Burges: The MNIST database of handwritten digits.

https://www.lri.fr/~marc/Master2/MNIST_doc.pdf

[3] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton: The CIFAR-10 dataset.

https://www.cs.toronto.edu/~kriz/cifar.html

[4] Alex Krizhevsky, Learning Multiple Layers of Features from Tiny Images, April 2009

https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[5] Jason Brownlee: A Gentle Introduction to the Rectified Linear Unit (ReLU), August 20, 2020.

https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[6] Eric W. Weisstein: Wolfram Mathworld – Hyperbolic Tangent
https://mathworld.wolfram.com/HyperbolicTangent.html

[7] Eric W. Weisstein: Wolfram Mathworld – Sigmoid Tangent
https://mathworld.wolfram.com/SigmoidFunction.html

CHAPTER 2: BACKPROPAGATION ALGORITHM

INTRODUCTION

The previous chapter explained the operation of a single artificial neuron. It covered how input values are multiplied by their respective weight parameters, summed together, and combined with a bias term. The resulting value, z, is then passed through a non-linear sigmoid function, which squeezed a neuron's output value $yy^$ between 0 and 1.

In this chapter, we form the smallest possible Feed Forward Neural Network (FFNN) model using only two neurons. While this is far from a Deep Neural Network (DNN), a simple NN with two neurons is sufficient to explain the Backpropagation algorithm, which is the focus of this chapter.

The goal is to demonstrate the training process and illustrate how the Forward Pass (computation phase) first generates a model output, $yy^$. The algorithm then evaluates the model's accuracy by computing the error term using Mean Squared Error (MSE). The first training iteration rarely, if ever, produces a perfect output. To gradually bring the training result closer to the expected value, the Backward Pass (adjustment and communication phase) calculates the magnitude and direction by which the weight values should be adjusted. We are using a supervised training process with a prelabeled test dataset, although it is not shown in Figure 2-1. Chapter Three covers the training datasets in detail.

After the training process is completed, we use a test dataset to evaluate the model's performance. Test dataset also contains input data and labels, but these labels are not used during training. Instead, after training is complete, the model is evaluated on the test dataset to measure its performance. At this phase, we measure how well the predictions from the training and test phases align. When the model produces the expected results on the test dataset, it can be taken into production.

FORWARD PASS

Figure 2-1 illustrates how neuron-a computes a weighted sum from three input values, adds a bias term, and produces a pre-activation value za. This value is then passed through the Sigmoid activation function. The output *yy*^a from neuron-a serves as an input for neuron-b, which processes it and generates the final model output *yy*^b. Since these computational steps were covered in detail in Chapter 1, we will not repeat them here.

As the final step of the Forward Pass, we apply the error function E to the model output. The error function measures how far model output *yy*^b is from expected value y. We use the Mean Squared Error (MSE), which is computed by subtracting the expected value from the model's output, squaring the result, and multiplying it by 0.5 (or equivalently, dividing by two).

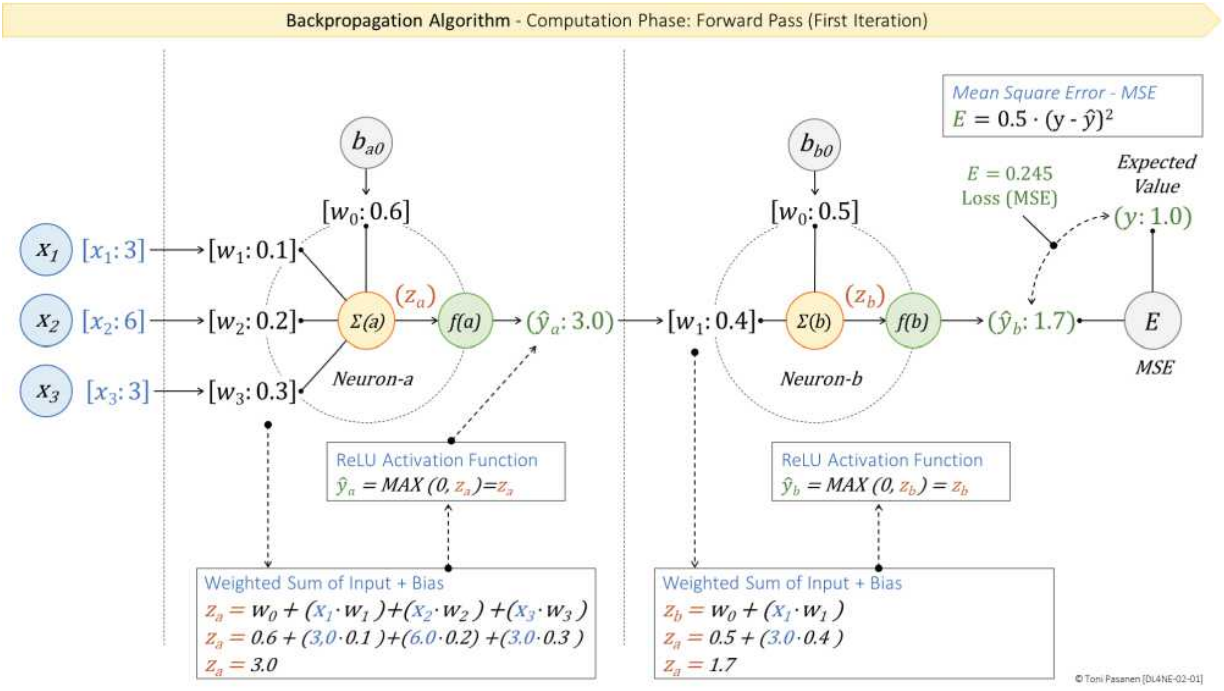


Figure 2-1: *An Overview of a Complete Forward Pass Process.*

On the right side of Figure 2-2, we have a two-dimensional error space. In this space, a symmetric parabolic curve visualizes the error function. The curve is centered at the expected value, which is 1.0 in our example. The horizontal axis represents the model output, $yy^$, and the vertical axis represents the error E. For instance, if the model prediction is 1.7, you can draw a vertical line from this point on the horizontal axis to meet the parabolic curve. In our case, this intersection shows an error term of 0.245. In real-life scenarios, the error landscape often has many peaks and valleys rather than a simple symmetric curve.

The Mean Squared Error (MSE) is a loss function that measures the difference between the predicted values and the expected values. It provides an overall error value for the model, which is

also called the loss or cost, indicates how far off the predictions are.

Next, the gradient is computed by taking the derivative of the loss function with respect to the model's weights. This gradient shows both the direction and the magnitude of the steepest increase in error. During the Backward Pass, the algorithm calculates the gradient for each weight. By moving in the opposite direction of the gradient (using a method called Gradient Descent), the algorithm adjusts the weights to reduce the loss. This process is repeated many times so that the model output gradually becomes closer to the expected value. The Backward pass process is explained right after the Learning Rate section.

The following sections will cover the processes and computations performed during the Backward Pass.

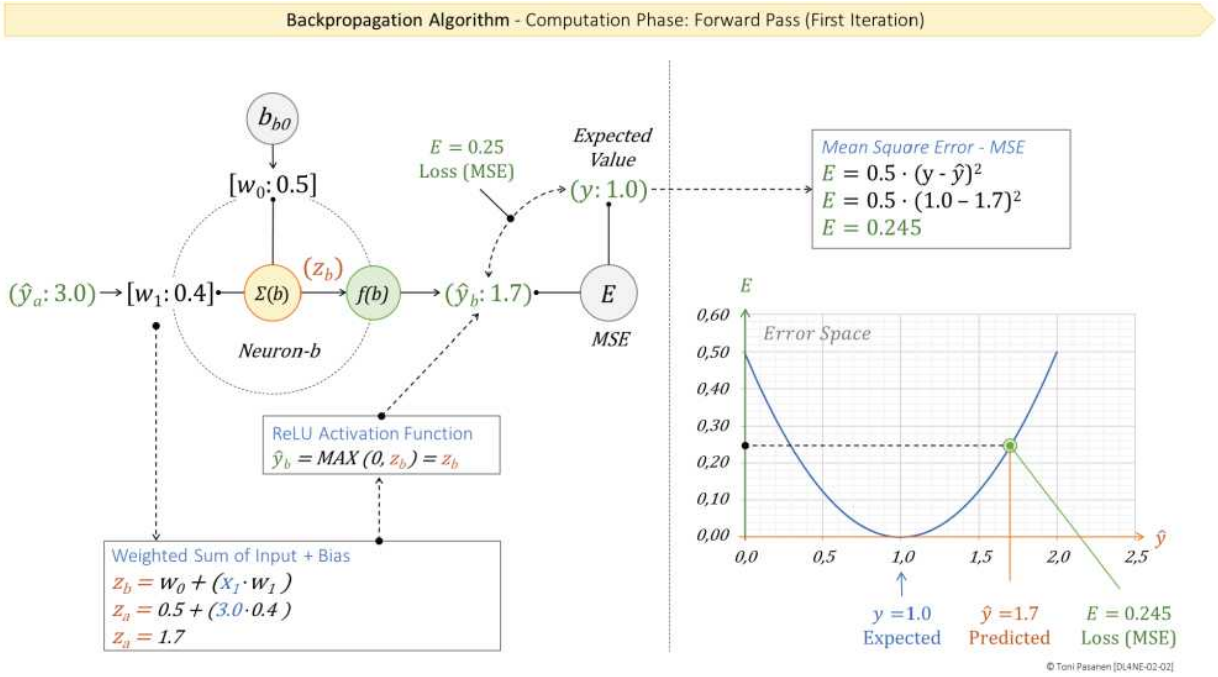


Figure 2-2: *Mean Square Error.*

LEARNING RATE

Besides determining the direction in which the error should be reduced, the process also needs to know the size of each adjustment step. This is defined by the Learning Rate. The Learning Rate value affects how much the weights are adjusted in response to the gradient during each iteration of the Backward Pass. A small Learning Rate leads to small, gradual changes, which may result in slower training but a more stable convergence. On the other hand, a large Learning Rate can speed up training by making larger adjustments, yet it might overshoot the optimal values and cause instability. Therefore, choosing the right Learning Rate is crucial for effective and efficient training. This is illustrated in the Figure 23. We will get back to Learning Rate in the Backward Pass section.

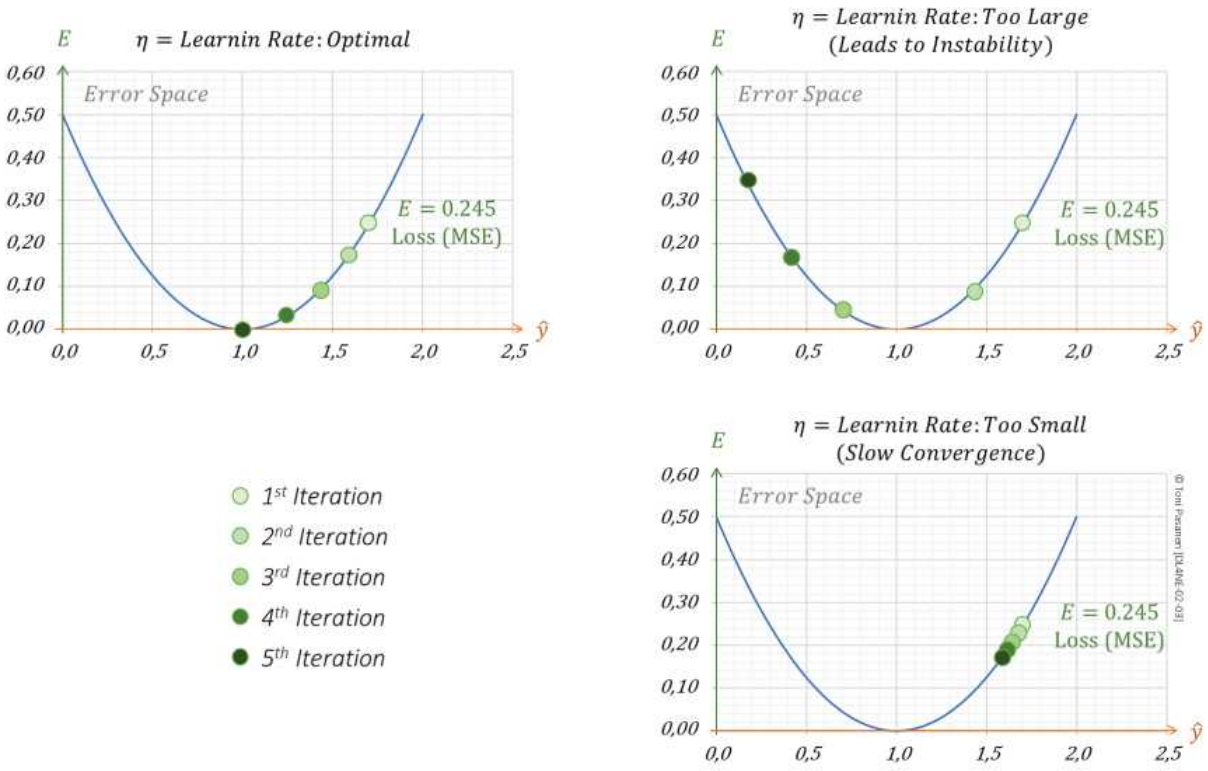


Figure 2-3: *Learning Rate.*

BACKWARD PASS

The Forward Pass produces the model output $\hat{y}$, which is then used to compute the model error E. The closer $\hat{y}$ is to the expected value y, the smaller the error, indicating better model performance. The purpose of the Backward Pass, as part of the Backpropagation algorithm, is to adjust the model's weight parameters during training in a direction that gradually moves the model's predictions closer to the expected values y.

In Figure 2-4, the model's output $\hat{y}$b depends on the weighted sum zb of neuron-b. This weighted sum zb, in turn, is calculated by multiplying an input value ya by its associated weights w1. The same process applies to neuron-a. Backpropagation algorithm cannot directly modify the results of an activation function or the weighted sum itself. Nor can it alter the input values directly. Instead, it calculates weight adjustments, which are then used to update the model's weights.

Figure 2-4 illustrates this dependency chain and provides a high-level overview of how the Backpropagation algorithm determines weight adjustments. The following sections will explain this process in detail.

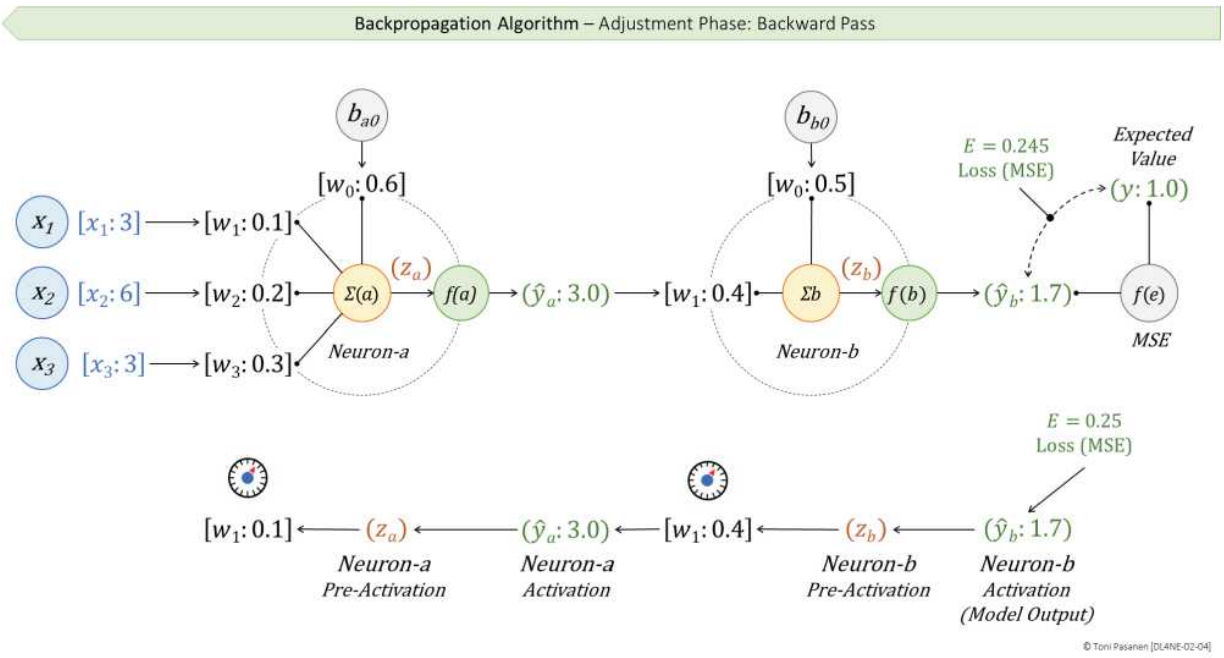


Figure 2-4: *Backpropagation Overview: Backward Pass Dependency Chain.*

The somewhat crowded Figure 2-5 illustrates the components of the backpropagation algorithm, along with their relationships and dependencies. The figure consists of three main blocks. The rightmost block depicts the calculation of the error function. The middle and left blocks outline the steps for defining and adjusting new weight values. The complete backward pass process is explained next in detail, one step at a time.

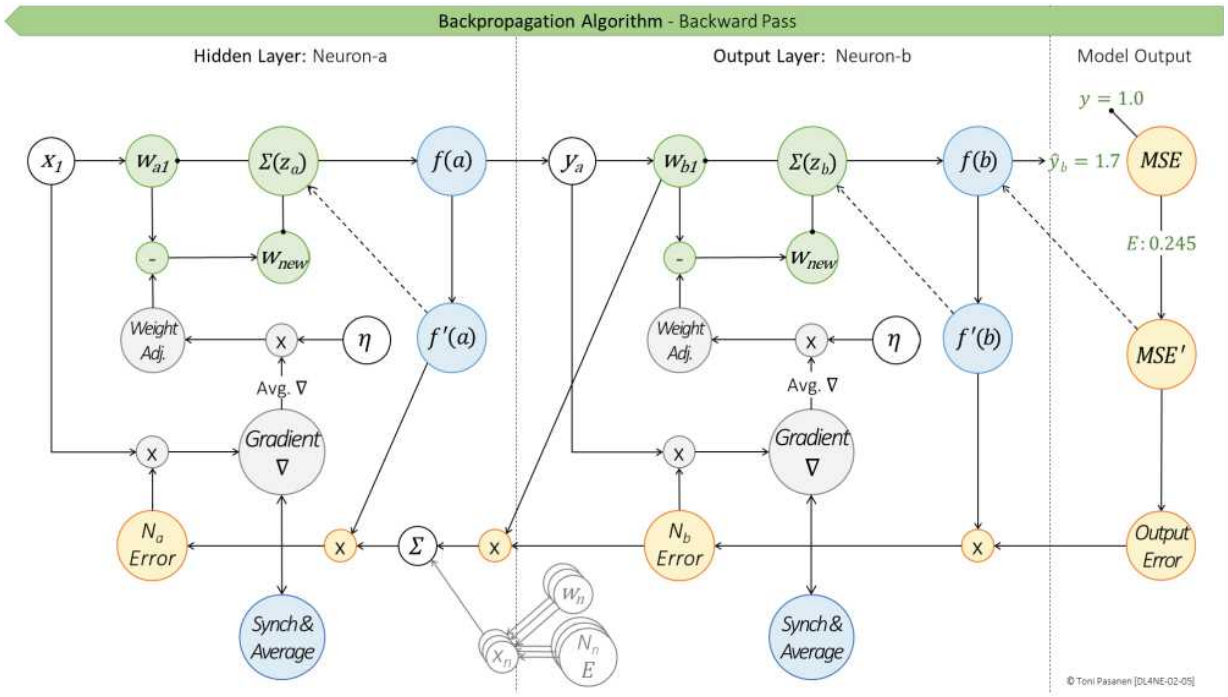


Figure 2-5: *The Backward Pass Overview.*

Partial Derivative for Error Function – Output Error

The goal of training a model is to minimize the error, meaning we want *yy*^b (the model's prediction/output) to get as close as possible to y (the expected value).

After computing the error E (=0.245), we compute the partial derivative of the error function with respect to yb (=1.7), which shows how small changes in *yy*^b affects the error E. A derivative is called partial when one of its input values is held constant (i.e., not adjusted by the algorithm). In our example, the expected value y is constant input. The result of the partial derivative of the error function indicates how the predicted output *yy*^b should change to minimize the model's error.

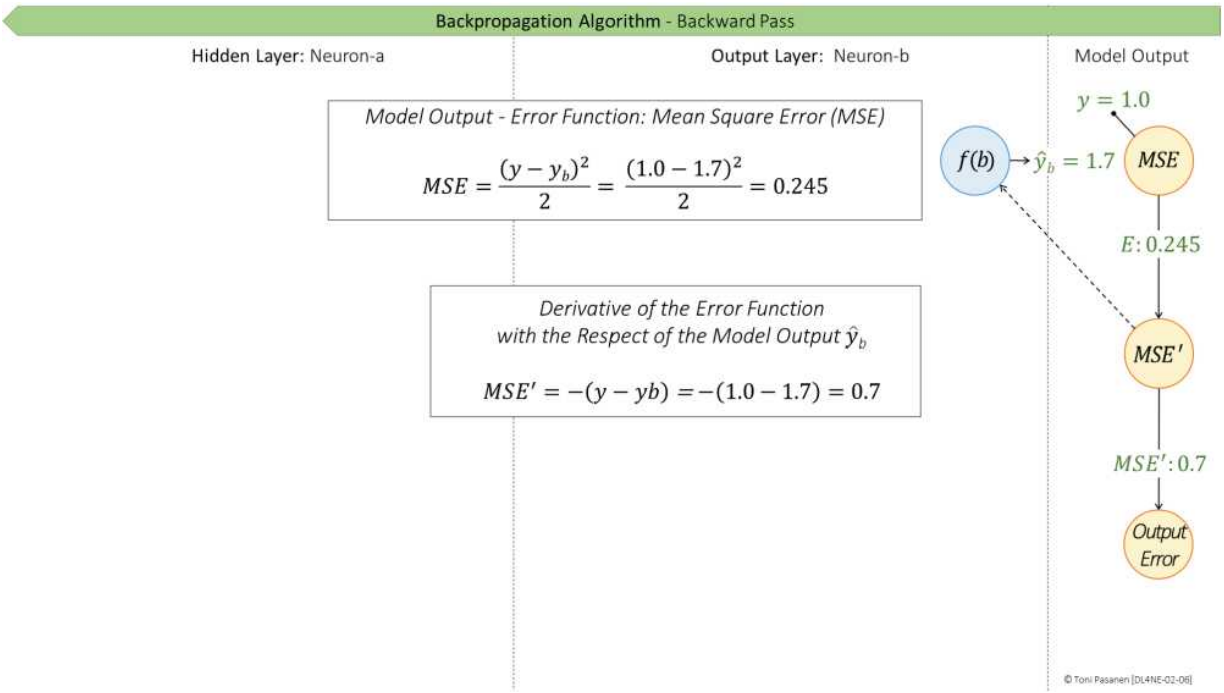
We use the following formula for computing the derivative of the error function:

$MSE' = -(yy - yyyy)$

$MSE' = -(1.0 - 1.7)$

$MSE' = 0.7$

Since the model output yb =1.7 is too high, the positive gradient suggests that it should be lowered by 0.7, which is the derivative of the error function (MSE'). This makes perfect sense because by subtracting the MSE's 0.7 from the model output yb = 1.7, we obtain 1.0, which matches the expected value.



Partial Derivative for the Activation Function

After computing the output error, we calculate the derivative of the activation function f(b) with respect to zb. Neuron-b uses a

ReLU activation function, which states that if the function's output is greater than 0, the derivative is 1; otherwise, it is 0. In our case, the result of the activation function f(b)=1.7, so the derivative is 1.

$ff'(yy)$ = ^

1, if $ff(yy)$ >0

0, if $ff(yy) \leq 0$

Error Term for Neurons

The error term for neuron-b is calculated by multiplying the partial derivative of the error function MSE' = 0.7, by the derivative of the neuron's activation function $ff'(yy)$= 1.0. This means we propagate the model's error backward using it as a base value for finetuning the model accuracy (i.e., refining new weight values). This is why the term backward pass fits perfectly for the process.

Error term (Enb) for Neuron-b = MSE' $\cdot$ $ff'(yy)$ = 0.7 $\cdot$ 1 = 0.7

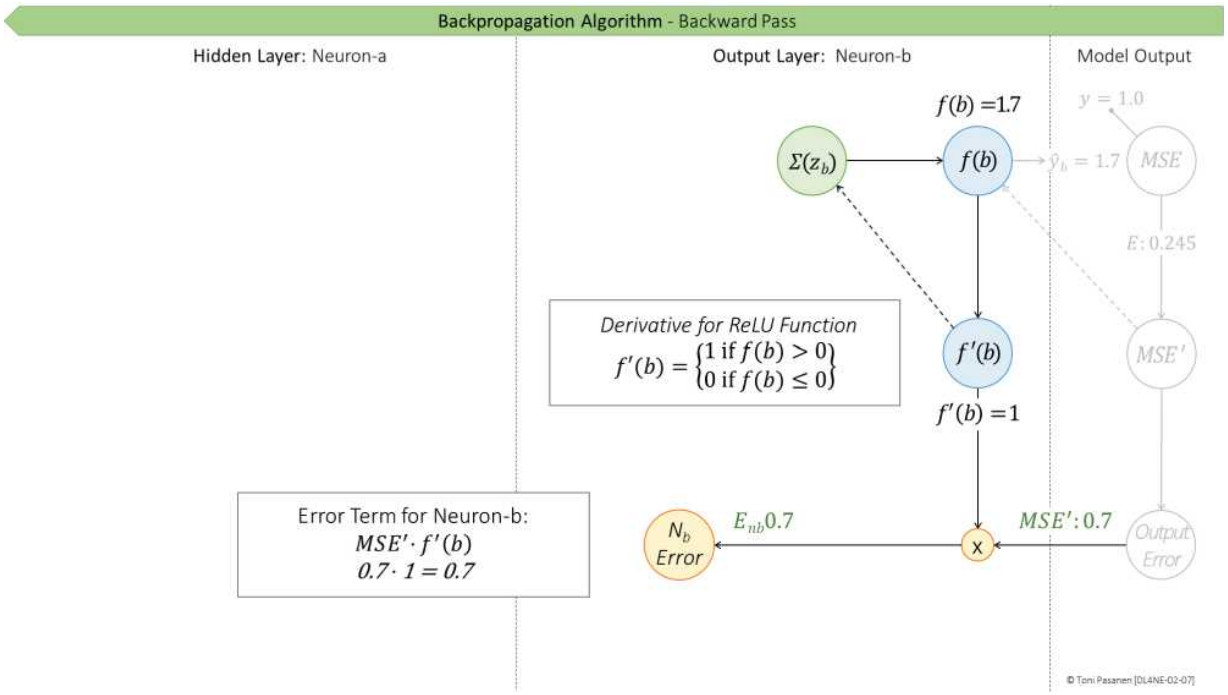


Figure 2-7: *The Backward Pass – Error Term for Neuron-b.*

After computing the error term for neuron-b, the backward pass moves to the preceding layer, the hidden layer, to calculate the error term for neuron a. First, the process computes a weighted sum of $w \cdot E$ across all connected neurons in the next layer, output layer in our example. This sum is then multiplied by the derivative of the activation function, $ff'(a)$. Since neuron-a is only connected to neuron-b, its error term is calculated as $w1 \cdot Enb \cdot ff'(a)$, resulting error term for neuron-a, $Ean = 0.4 \cdot 0.7 \cdot 1 = 0.28$.

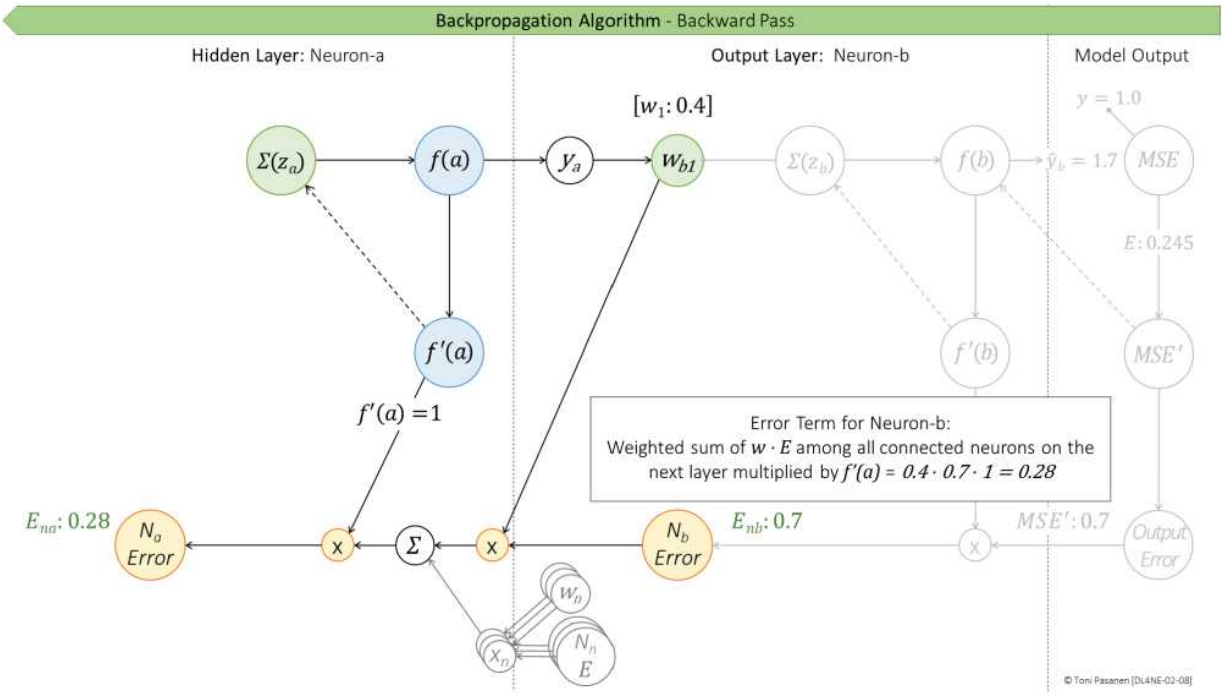


Figure 2-8: *The Backward Pass – Error Term for Neuron-a.*

Gradient Calculation

After computing the error terms for all neurons in every layer, the algorithm simultaneously calculates gradients for all weight parameters. Each gradient is determined by multiplying the input value by the corresponding error term.

In our example, the gradient for weight $wa1$, which connects input $x1$ to neuron-a, is calculated by multiplying the input value $x1$ (=3.0) by the error term Ena of neuron-a (= 0.28), resulting in a gradient of 0.84. Similarly, the gradient for weight $wb1$ in neuron-b is computed by multiplying the output y (=3.0) of the activation function from neuron-a by the error term Enb of neuron b (=0.7), yielding a gradient of 2.1.

If the test dataset is divided across multiple GPUs, gradients must be synchronized before computing the actual weight-based adjustment values. Each GPU sums all received gradients, including its own. The sum is then averaged by dividing it by the number of GPUs. This process is explained in detail in Chapter 8. Next, the GPUs synchronize these averaged gradients to ensure that each one uses the same values when calculating the final weight adjustments. This process is part of a data parallelization strategy, where the training dataset is too large to fit into a single GPU's memory and is split into micro-batches. Each GPU processes its micro-batches using the same model with the same parameters.

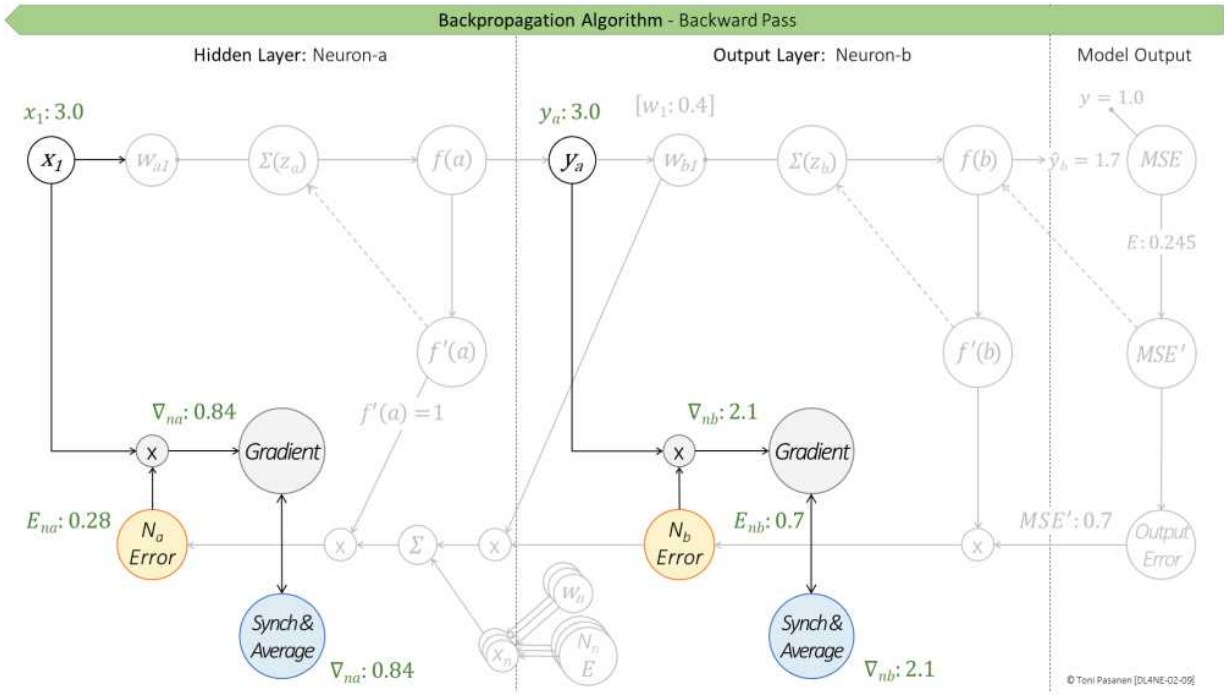


Figure 2-9: *The Backward Pass – Gradient for Neurons.*

Weight Adjustment

The weight adjustment value is computed by multiplying the gradient, averaged in our example, by the learning rate η. We use a learning rate of 0.012. This results in a weight adjustment of 0.042 for weight wa1 and 0.105 for weight wb1.

The weight adjustment values are then subtracted from the initial weights. This yields an updated weight of 0.058 (0.1-0.042) for wa1 and 0.295 (0.4-0.105) for wb1.

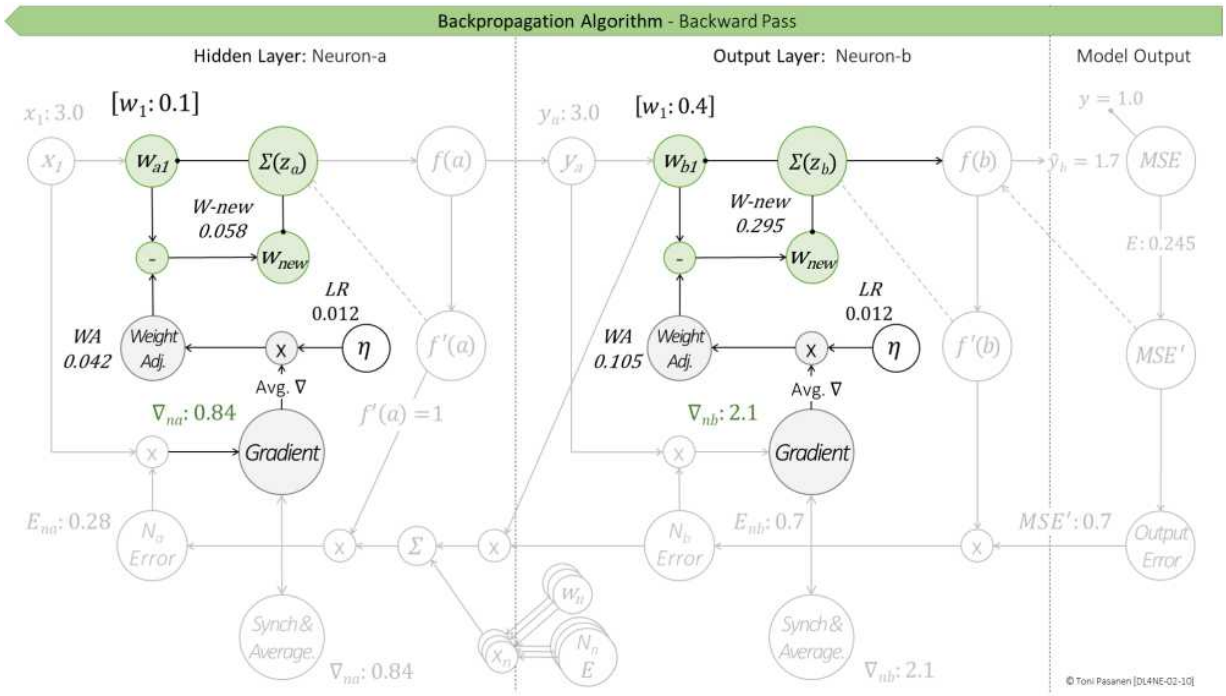


Figure 2-10: *The Backward Pass – Compute New Weight Values.*

THE SECOND ITERATION - FORWARD PASS

After updating all the weight values, including those associated with biases, the backpropagation process begins the second iteration of the forward pass. As shown in Figure 2-11, the model output $yy\hat{}b$ = 1.28 is very

close to the expected value y = 1.0. The new MSE = 0.007 is significantly lower than the initial MSE = 0.245 computed in the first iteration.

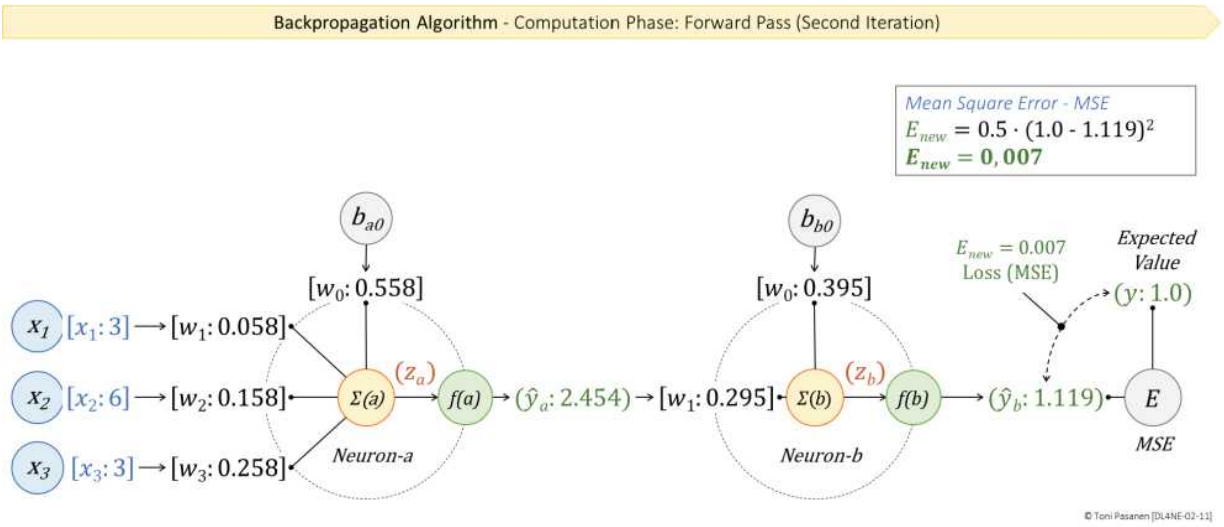


Figure 2-11: *The Second Iteration of the Forward Pass.*

In Figure 2-12, we have two 2-dimensional error spaces. Using the initial weight values, the model output is 1.7, resulting in an MSE of 0.245. After adjusting the weights, the model prediction is 1.119, reducing the MSE to 0.007.

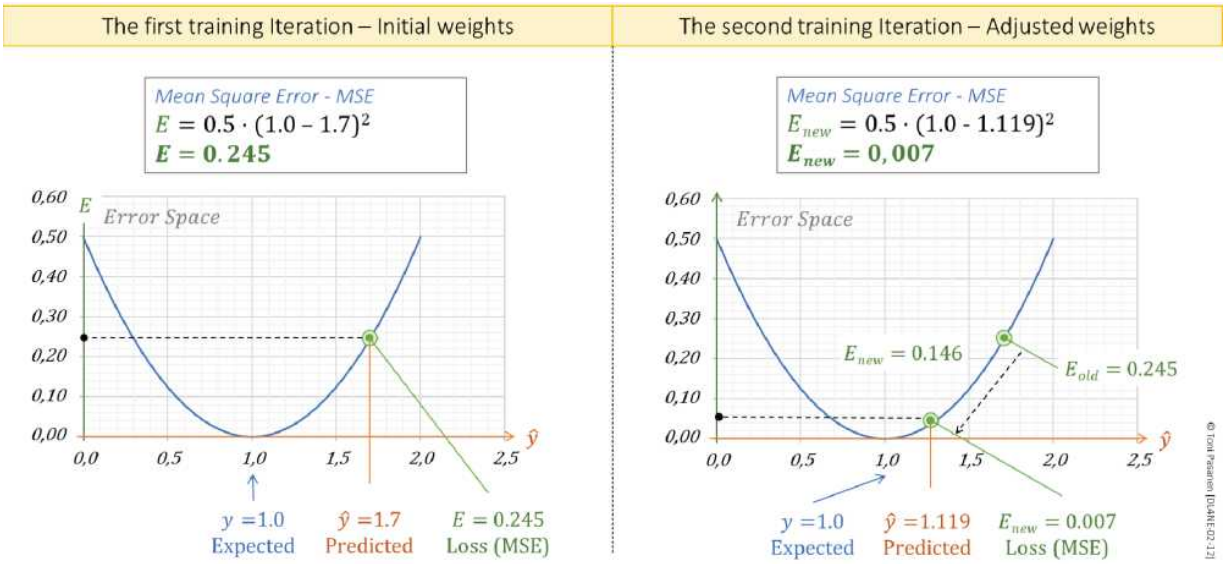


Figure 2-12: *Results Comparison.*

NETWORK IMPACT

In figure 2-13 we have a fully-connected Feed Forward Neural Network (FFNN) with four layers; input layer, two hidden layers, and output layer. Training data set is split into two batches, A and B, which are processed by GPU-A and GPU-B.

After computing a model prediction during the forward pass, the backpropagation algorithm begins the backward pass by calculating the gradient (output error) for the error function. Once computed, the gradients are synchronized between the GPUs. The algorithm then averages the gradients, and the process moves to the preceding layer. Neurons in the preceding layer calculate their gradient by multiplying the weighted sum of their connected neurons' averaged gradients and connected weight with the local activation function's partial derivative. These neuron-based gradients are then synchronized over connections (the process is explained in detail in chapter 14). Before the process moves to the preceding layer, gradients are averaged. The backpropagation algorithm executes the same process through all layers.

If packet loss occurs during the synchronization, it can ruin the entire training process, which would need to be restarted unless snapshots were taken. The cost of losing even a single packet could be enormous, especially if training has been ongoing for several days or weeks. Why is a single packet so

important? If the synchronization between the gradients of two parallel neurons fails due to packet loss, the algorithm cannot compute the average, and the neurons in the preceding layer cannot calculate their gradient. Besides, if the connection, whether the synchronization happens over NVLink, InfiniBand, Ethernet (RoCE or RoCEv2), or wireless connection, causes a delay, the completeness of the training slows down. This causes GPU under-utilization which is not efficient from the business perspective.

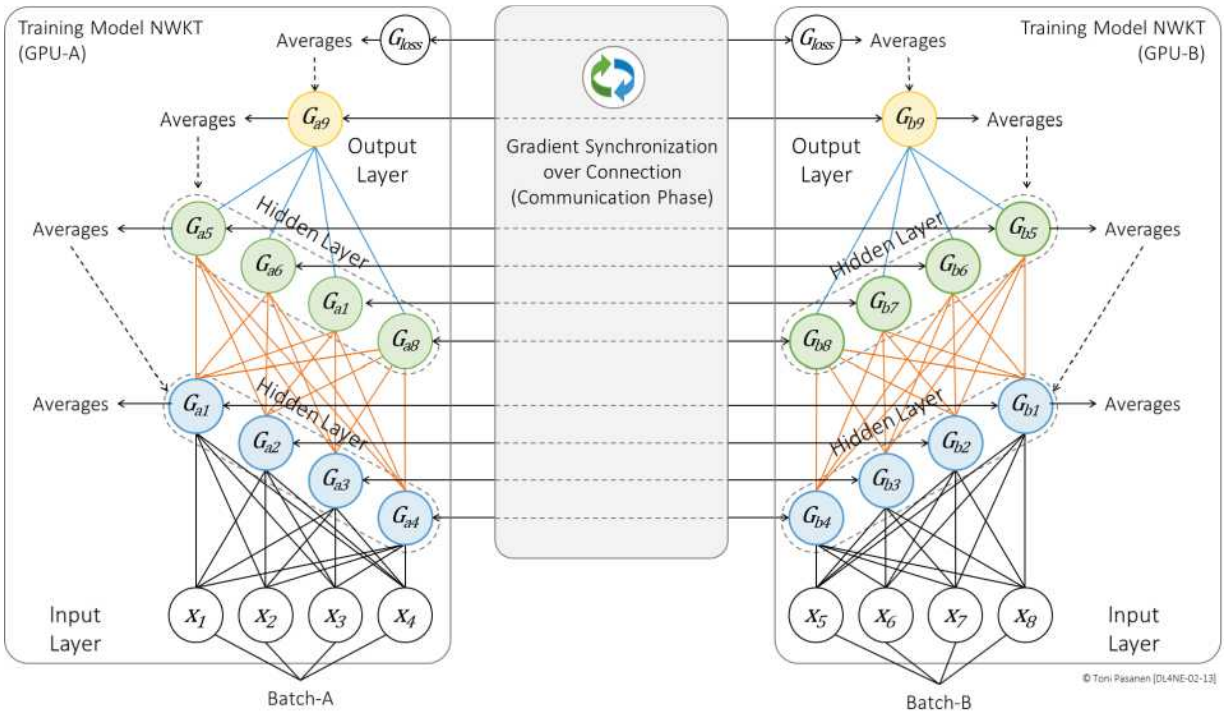


Figure 2-13: *Backward Pass – Gradient Synchronization and Averaging.*

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016, November 18). Deep Learning. MIT Press.

https://www.deeplearningbook.org/

[3] Nielsen, M. (2015). Neural Networks and Deep Learning – Chapter 2: How the Backpropagation Algorithm Works. Determination Press.

http://neuralnetworksanddeeplearning.com/chap2.html

[4] LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (2012). Efficient BackProp. In Neural Networks: Tricks of the Trade (pp. 9–50). Springer.

https://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[5] Stanford University (CS231n). (2025, April 10). Lecture 4: Neural Networks and Backpropagation.

https://cs231n.stanford.edu/slides/2025/lecture_4.pdf

[6] MIT Vision Book. (2024, April 16). Chapter 14 – Backpropagation. In Foundations of Computer Vision. MIT Press.

https://visionbook.mit.edu/backpropagation.html

[7] Mazur, M. (2015, March 17). A Step by Step Backpropagation Example.

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

CHAPTER 3: MULTI-CLASS CLASSIFICATION

INTRODUCTION

This chapter explains the multi-class classification training process. It begins with an introduction to the MNIST dataset (Modified National Institute of Standards and Technology). Next, it describes how the SoftMax activation function computes the probability distribution over digit classes during the forward pass and how the model's weight parameters are updated during the backward pass to improve classification accuracy. Additionally, the chapter discusses the data parallelization strategy from a network perspective.

MNIST DATASET

We will use the MNIST dataset, which consists of grayscale images of handwritten digits, to demonstrate the training process. The MNIST dataset includes four binary files: a training set with 60,000 images and their corresponding labels, and a test set with 10,000 images and labels. Each image is 28×28 pixels in size.

The files are:

train-images-idx3-ubyte: contains the pixel values for the training images, along with metadata describing the file format.

train-labels-idx1-ubyte: contains the labels (digits 0–9)

corresponding to each image in the training set.

t10k-images-idx3-ubyte: contains the test images.

t10k-labels-idx1-ubyte: contains the labels for the test images.

Since there are ten possible digits (0–9), the output layer of the model uses ten neurons, each representing one digit class.

Before training begins, the labels for each image-label pair are one-hot encoded. This means that each label is transformed into a vector of ten elements: the correct class is represented by a 1 at its index position, and all other positions are set to 0. For example, if an image corresponds to the digit 8, the one-hot

vector would be [0, 0, 0, 0, 0, 0, 0, 0, 1, 0] (index 8 is set to 1, assuming 0-based indexing).

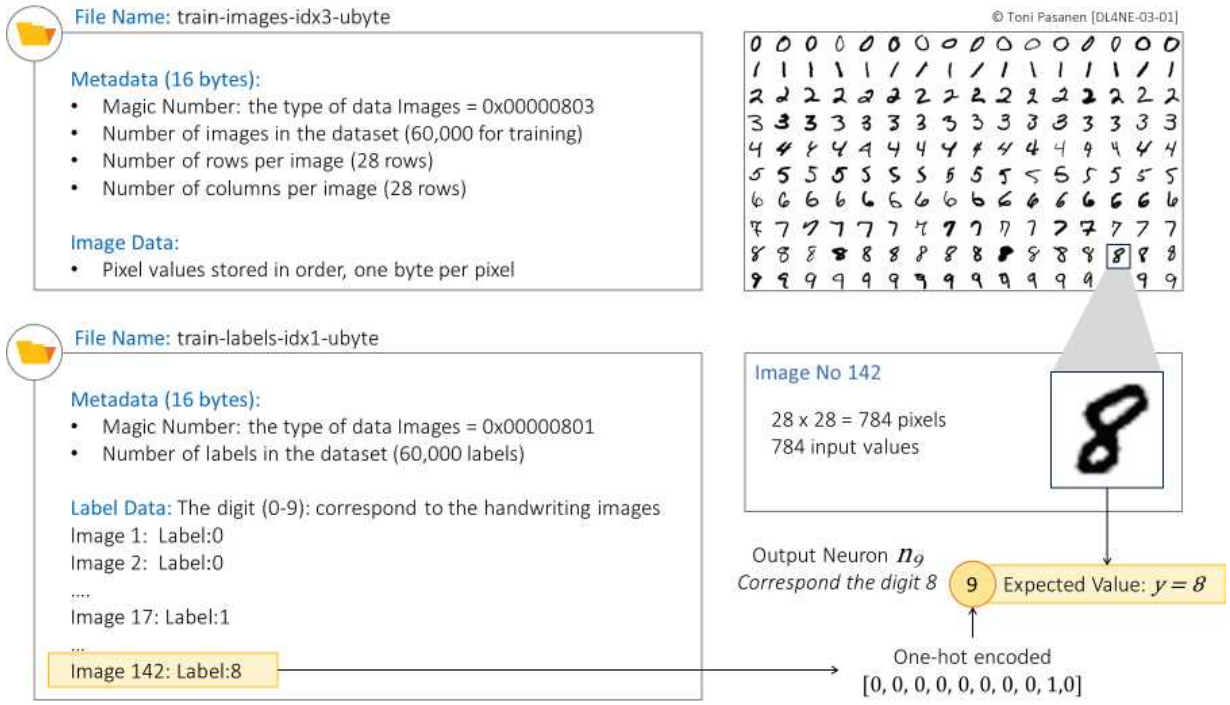


Figure 3-1: *Training Dataset & Labels– The MNIST Database.*

FORWARD PASS

Model Probability

Figures 3-2 and 3-3 illustrate the forward pass process for multi-class classification. The *Input Layer* flattens the 28 x28 pixel image into 784 input parameters, where each parameter represents the intensity of a pixel (0 = black, 255 = white). These 784 input values are then passed to all 128 neurons in the *Hidden Layer*. Each neuron in the hidden layer receives all 784 inputs, and each of these inputs is associated with a unique weight. Therefore, each of 128 neurons have 784 weight parameters, and total weight parameter count of hidden layer is 100 352.

In the hidden layer, each neuron computes the weighted sum (= *Matrix Multiplication*) of its inputs and then applies the ReLU activation function to the result. This process produces 128 activation values, one for each neuron in the hidden layer.

Next, these 128 activation values are fed into the *Output Layer*, which consists of 10 neurons (corresponding to the 10 possible classes for the MNIST dataset). Each output neuron is connected to all 128 activation values from the hidden layer. Therefore, the weight parameter counts in the output layer is 1280. Again, each neuron does matrix multiplication by computing a weighted sum of its inputs. The result of this calculation is called a logit.

In the output layer, the SoftMax activation function is applied to these logits. SoftMax first computes the exponential of each logit, *using Euler's number* e as the base. Then, it computes the sum of these exponentials, which in this example is 24.813. The probability for each class (denoted as $yy^$) is calculated by dividing each neuron's exponential by the sum of all exponentials.

In this example, the output neuron corresponding to class "5" produces the highest probability, meaning the model predicts the digit in the image is 5. However, since this prediction is incorrect in the first iteration, the model will adjust its weights during backpropagation.

In our model, we have 101,632 weight parameters. The number of bits used to store each weight parameter in a neural network depends on the numerical precision chosen for the model. The 32-bit floating point (FP32 – single precision) is the standard precision used, where each weight is represented using 32 bits (4 bytes). This format offers good precision but can be memory-intensive for large models. To reduce memory usage and increase speed, many modern hardware systems use 16-bit floating point (FP16 – half precision), where each weight is represented using 16 bits (2 bytes). There is also 64-bit floating point (FP64 – double precision), which uses 64 bits (8 bytes), providing more precision and a larger range than FP32, but at the cost of increased memory usage.

In our model, using FP32, the memory required for the weight parameters is 406,528 bytes (4 × 101,632).

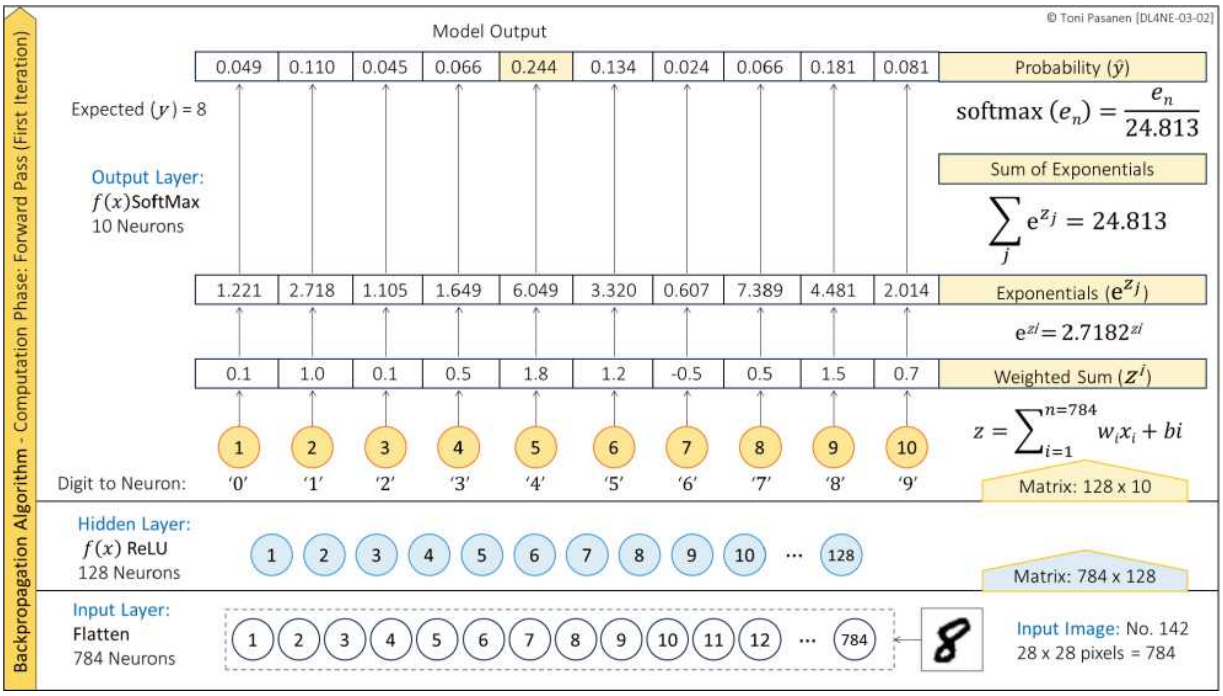


Figure 3-2: *Forward pass – Probability Computation.*

Cross-Entropy Loss

In our example, the highest probability value (0.244) is provided by neuron 5, though the expected value should be produced by neuron 9. Next, the algorithm computes the cross-entropy loss by taking the logarithm of the probability value for the expected neuron, as defined by the one-hot encoded label. In our example, the probability of the digit being 8, computed by neuron 9, is 0.181. The cross-entropy loss is calculated by taking the log of 0.181, resulting in 0.734.

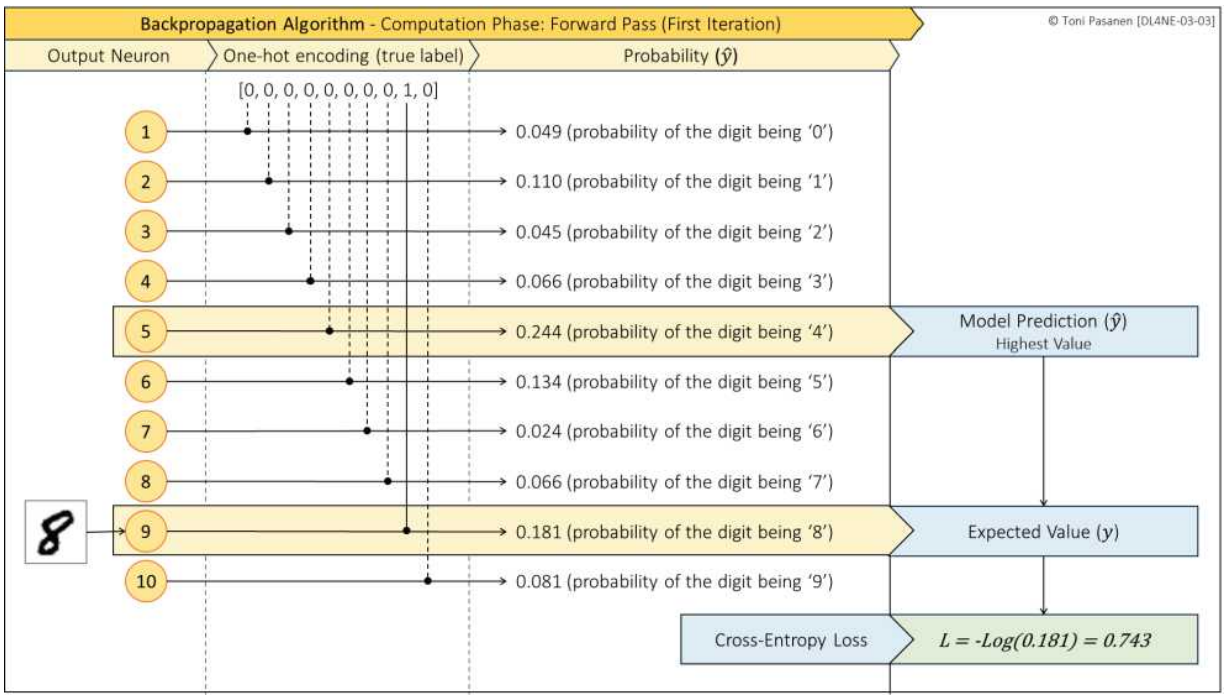


Figure 3-3: *Forward pass – Cross-Entropy Loss.*

BACKWARD PASS

Gradient Computing

The gradient for the neurons in the output layer is calculated by subtracting the ground truth values (from the one-hot encoded label) from the probabilities produced by the SoftMax function. This simplified gradient expression is a result of combining the SoftMax activation with the crossentropy loss, which cancels out more complex derivative terms. While the cross-entropy loss influences the training process, its derivative is implicitly included in this simplified gradient expression.

For neurons in the hidden layer, the gradient is computed by taking the weighted sum of the gradients from the connected output neurons. This sum is then multiplied by the derivative of the hidden neuron's activation function (such as ReLU). The formula for this computation is shown in Figure 3-4.

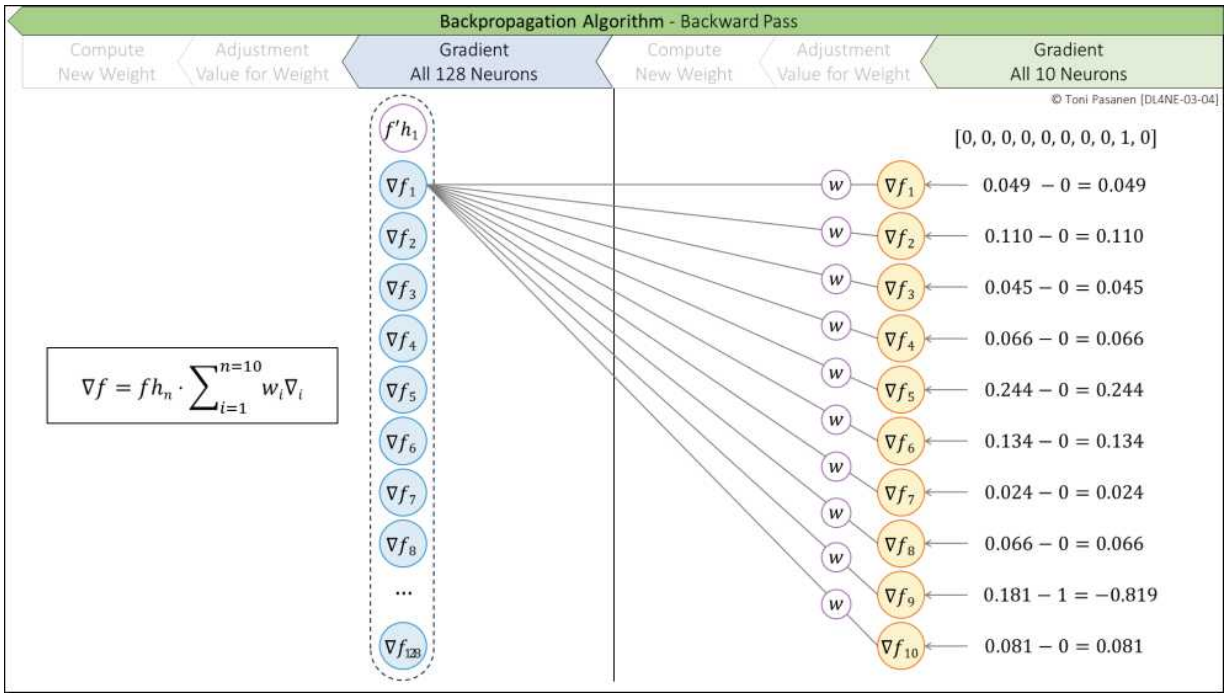


Figure 3-4: *Backward pass - Gradient Calculation.*

Weight Adjustment Values

After calculating the gradients for all neurons, the backpropagation algorithm determines the weight adjustments. While this process was explained in the previous chapter, let's briefly recap it here. Each weight adjustment is computed by multiplying the gradient of the neuron it connects to (the downstream neuron) by the input that passed through that weight during the forward pass. This product represents the gradient of the weight. The actual adjustment is then calculated by multiplying this gradient by the learning rate — a shared hyperparameter that controls how much the weight is updated during training.

Figure 3-5 illustrates the computation from two perspectives: neuron 9 in the output layer and neuron 1 in the hidden layer.

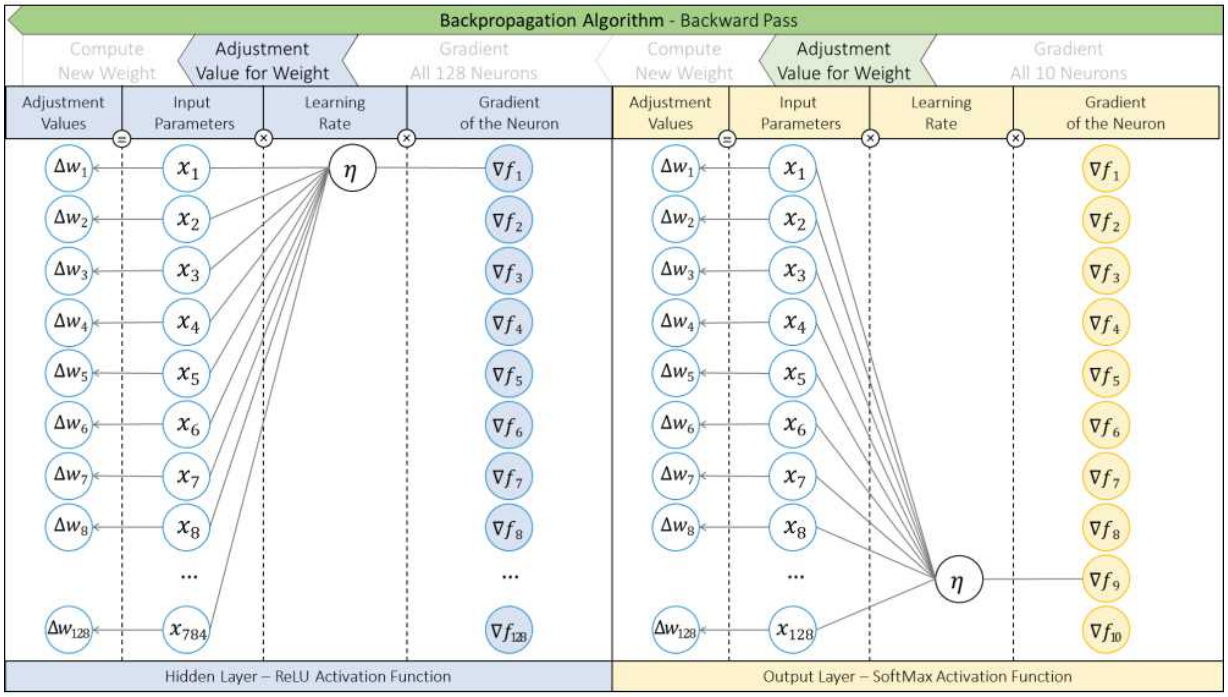


Figure 3-5: *Backward Pass - Weight Adjustment Value.*

Weight Update

Figure 3-6 depicts how the new weight value is obtained by adding the adjustment value to the initial weight value.

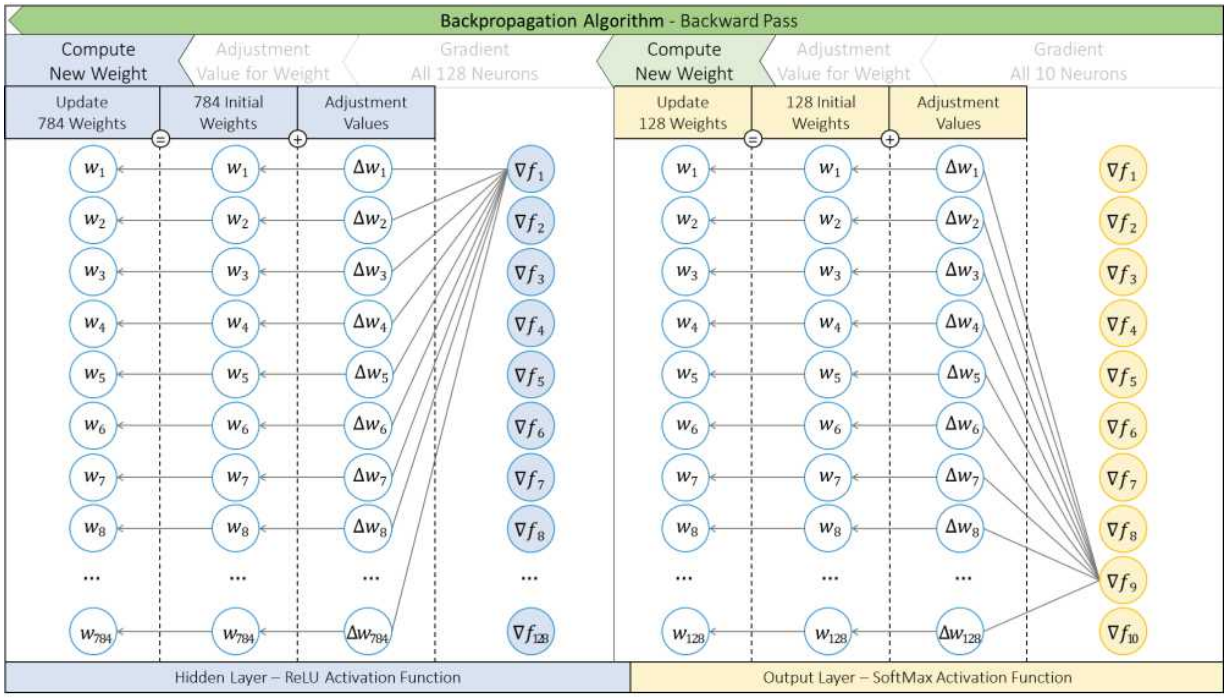


Figure 3-6: *Backward Pass – Computing New Value for the Weight Parameter.*

So far, we have explored how the backpropagation algorithm works in multi-class classification using a single GPU. In this section, we examine the scenario where the input dataset exceeds the memory capacity of a single GPU. To handle this, we adopt a data parallelization strategy that splits the training data across multiple GPUs. We also analyze the strategy from a network utilization standpoint.

In Figure 3-7, the training data is divided into mini-batches. The first half of the mini-batches is stored in system memory (DRAM-1) on Server-1, and the second half in system memory (DRAM-2) on Server-2. This setup illustrates a common situation: when the data cannot fully fit into a GPU's VRAM, inactive mini-batches remain in system memory and are

transferred as needed. Meanwhile, the active mini-batch and the model weights are loaded into the GPU's VRAM.

In our example, each mini-batch contains **64 grayscale images**, each of size **32 × 32 pixels**, meaning **1,024 input features per image.** The first hidden layer contains **128 neurons**, and the output layer contains **10 neurons.**

The total number of weight parameters is as follows:

- **Input to hidden layer:**

64 images × 128 neurons × 1,024 inputs = **8,388,608 weights**

- **Hidden to output layer:**

128 neurons × 10 neurons = **1,280 weights**

- **Bias weights:**

128 (hidden layer) + 10 (output layer) = **138 bias values**

Once the forward pass (computation phase) is complete, the backward pass begins. In this phase, gradients are computed for all neurons using the backpropagation algorithm.

To synchronize gradients across GPUs, we use the All-Reduce collective communication model (detail explanation in chapter 14), which aggregates gradients from each GPU and ensures that all GPUs have consistent copies of the model. In our multi-server setup, this synchronization takes place over the network using Remote Direct Memory Access (RDMA), a mechanism that allows one server's GPU to access another server's GPU memory

directly, bypassing CPU intervention and avoiding the traditional network stack. This form of RDMA (typically via RoCE or InfiniBand, explained in detail in chapter 9) is essential for minimizing latency and maximizing throughput across the cluster.

During gradient synchronization, the GPU's network interface controller (NIC) transmits data at line rate, often resulting in close to 100% link utilization. Once synchronization is complete, each GPU averages the gradients and computes updated weight values.

After the weights are updated, the next mini-batch is loaded into VRAM and training continues. During the compute-intensive forward and backward passes, network utilization is low, as most operations are local

to the GPU. When all GPUs are within the same server, high-speed interconnects like NVLink are typically used to facilitate fast GPU-to-GPU communication. We will cover intra-server memory transfers and their performance impact in a later chapter.

Given that model training can span days to weeks, it is critical that interGPU communication, particularly across servers, is lossless and capable of sustaining line-rate performance. To protect against training loss from network issues, regular checkpoints (snapshots of model weights) should be taken. Even a single dropped packet during gradient exchange could cause the model to diverge or the job to fail, requiring a complete restart.

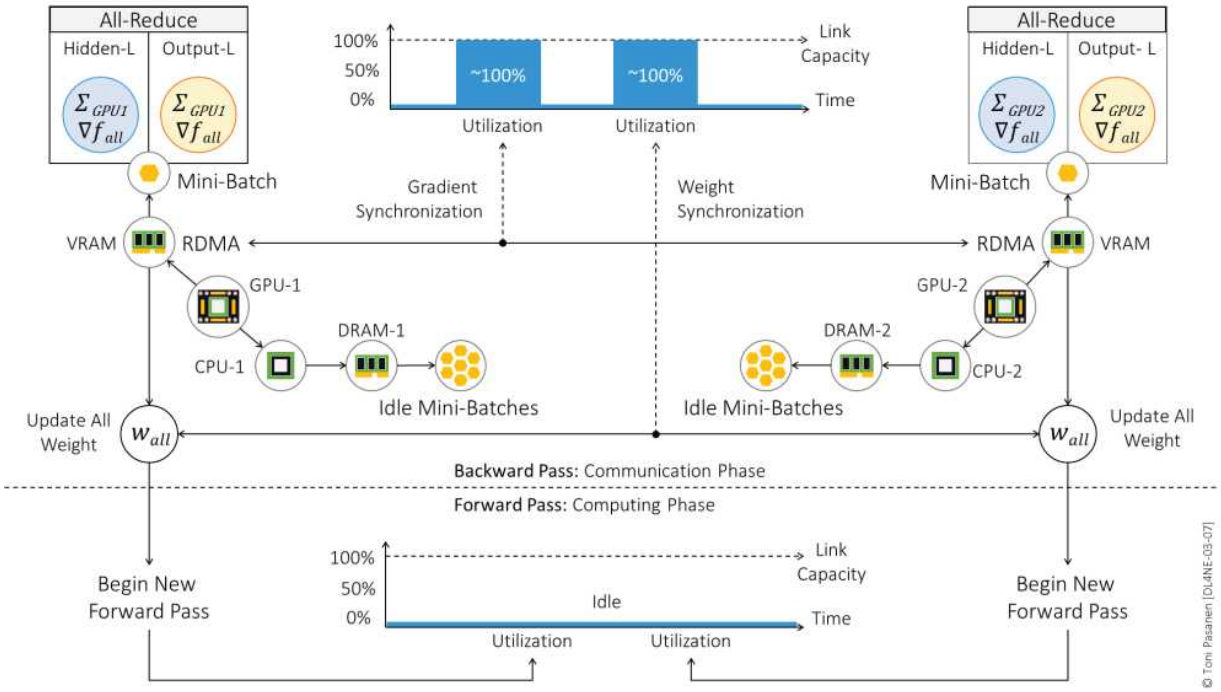


Figure 3-7: *Gradient Synchronization and Network Utilization.*

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Yann LeCun, Corina Cortes, Christoper J.C. Burges: The MNIST database of handwritten digits. October 2010.

https://www.lri.fr/~marc/Master2/MNIST_doc.pdf

[3] Lima, M. D. P., Giraldi, G. A., & Miranda Junior, G. F. (2023, November 10). Image Classification Using Combination of Topological Features and Neural Networks. arXiv preprint arXiv:2311.06375.

https://arxiv.org/abs/2311.06375

CHAPTER 4: CONVOLUTIONAL NEURAL NETWORKS

INTRODUCTION

The previous chapter explained how Feed-forward Neural Networks (FNNs) can be used for multi-class classification of 28 x 28 pixel handwritten digits from the MNIST dataset. While FNNs work well for this type of task, they have significant limitations when dealing with larger, high-resolution color images.

In neural network terminology, each RGB value of an image is treated as an input feature. For instance, a high-resolution 600 dpi RGB color image with dimensions 3.937 x 3.937 inches contains approximately 5.58 million pixels, resulting in roughly 17 million RGB values.

If we use a fully connected FNN for training, all these 17 million input values are fed into every neuron in the first hidden layer. Each neuron must compute a weighted sum based on these 17 million inputs. The memory required for storing the weights depends on the numerical precision format used. For example, using the 16-bit floating-point (FP16) format, each weight requires 2 bytes. Thus, the memory requirement per neuron would be approximately 32 MB. If the first hidden layer has 10,000 neurons, the total memory required for storing the weights in this layer would be around 316 GB.

In contrast, Convolutional Neural Networks (CNNs) use shared weight matrices called kernels (or filters) across all neurons within a convolutional layer. For example, if we use a 3 x 3 kernel, there are only 9 weights per color channel. This reduces memory usage and computational costs significantly during both the forward and backward passes.

Another limitation of FNNs for image recognition is that they treat each pixel as an independent unit. As a result, FNNs do not capture the spatial relationships between pixels, making them unable to recognize the same object if it shifts within the frame. Additionally, FNNs cannot detect edges or other important features. On the other hand, CNNs have a property called translation invariance, which allows the model to recognize patterns even if they are slightly shifted (small translations along the x and y axes). This helps CNNs classify objects more accurately. Furthermore, CNNs are more robust to minor rotations or scale changes, though they may still require data augmentation or specialized network architectures to handle more complex transformations.

CONVOLUTION LAYER

Convolution Process

The purpose of the convolution process is to extract features from the image and reduce the number of input parameters before passing them through fully-connected layers. The convolution operation uses a shared weight matrix called kernels or filters, which are shared across all neurons within a convolutional layer. In this example, we use the Prewitt operator, which consists of two 3 x 3 kernels with fixed weight values for detecting vertical and horizontal edges.

In the first step, these two kernels are positioned over the first region of the input image, and each pixel value is multiplied by the corresponding kernel weight. Next, the process computes the weighted sum, z=0, and the result is passed through the ReLU activation function. The resulting activation values, $ff\ (zz)=0$, contribute to the neuron-based output channels.

Since our input image is a grayscale image without color channels (unlike an RGB image), it has only one input channel. By using two kernels, we obtain two output channels: one for detecting vertical edges and the other for detecting horizontal edges. The formula for calculating the size of the output channel:

$$Height = (Image\ h - Kernel\ h)/Stride + bias = (6\text{-}3)/1 + 1 = 4$$

$Width = (Image\ w - Kernel\ w)/Stride + bias = (6\text{-}3)/1 + 1 = 4$

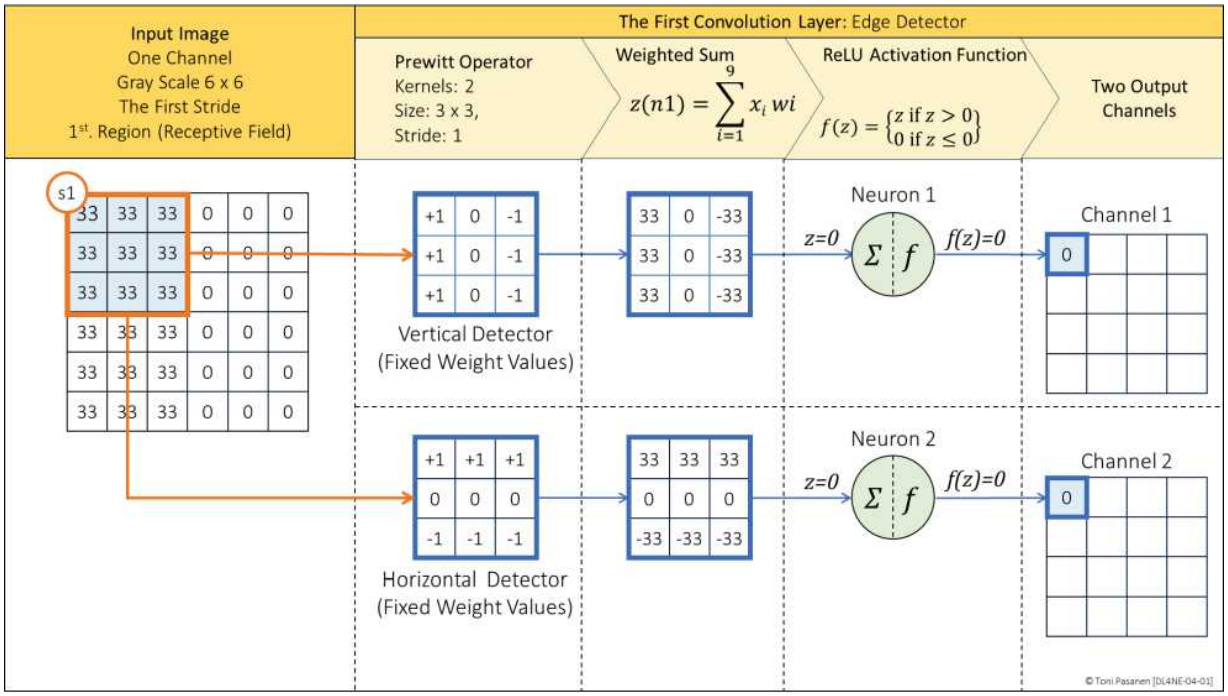


Figure 4-1: *Convolution Layer – Stride One.*

After calculating the first value for the output channel using the image values in the first region, the kernel is shifted one step to the right (stride of 1) to cover the next region. The convolution process calculates the weighted sum based on the values in this region and the weights of the kernel. The result is then passed through the ReLU activation function. The output of the ReLU activation function differs for the first output channel, it is $ff(z)=99$; for the second output channel, it is $ff(z)=0$.

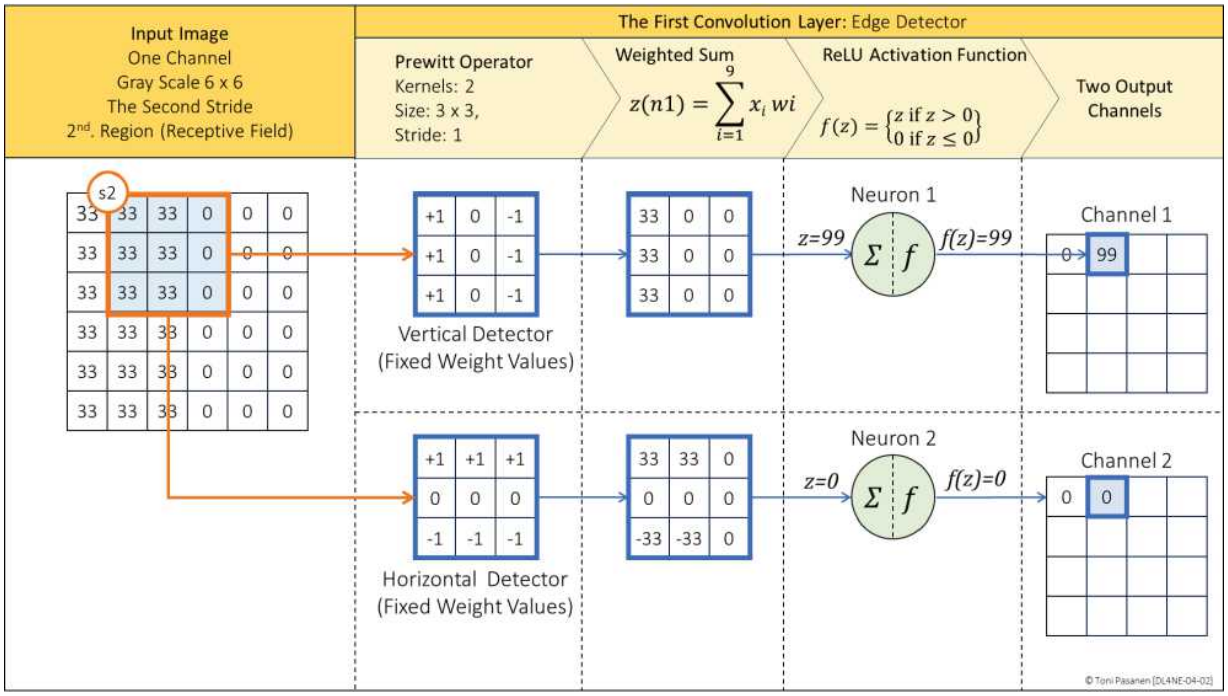


Figure 4-2: *Convolution Layer – Stride Two.*

Figure 4-3 depicts the fifth stride.

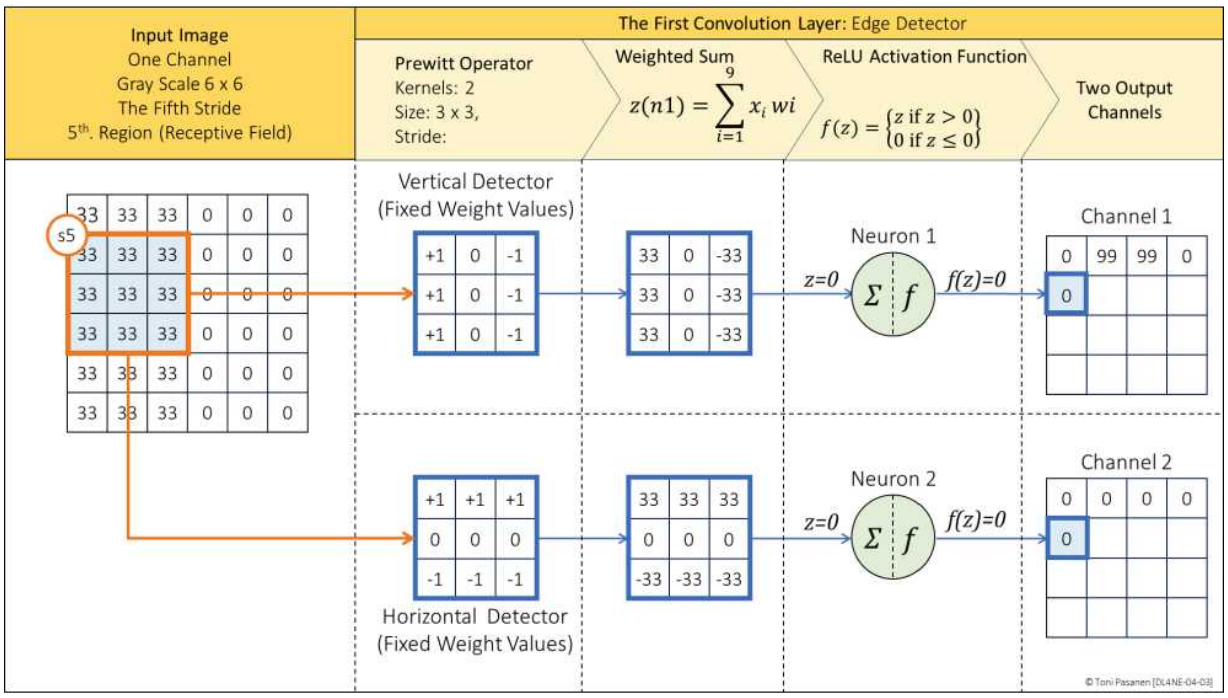


Figure 4-3: *Convolution Layer – Stride Five.*

The sixteenth stride, shown in Figure 4-4, is the last one. Now output channels one and two are filled.

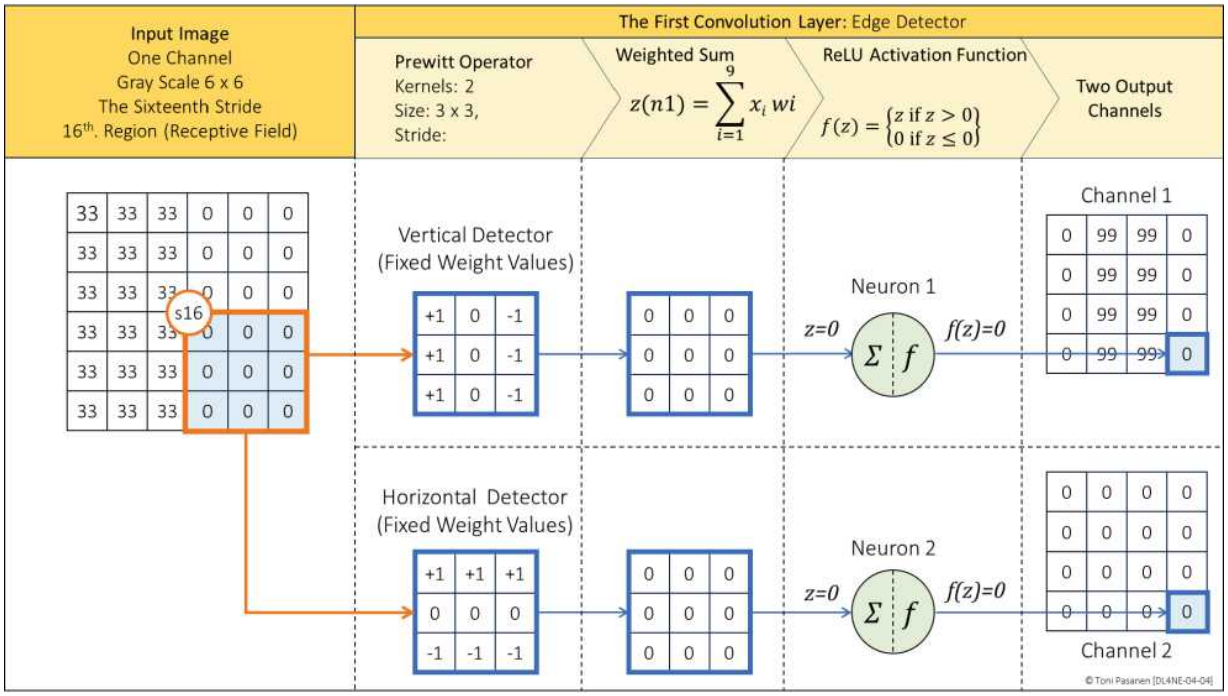


Figure 4-4: *Convolution Layer – Stride Sixteenth.*

Figure 4-5 shows how the convolution process found one vertical edge and zero horizontal edge from the input image. The convolution process produces two output channels, each with a size of 4 × 4 pixels, while the original input image was 6 × 6 pixels.

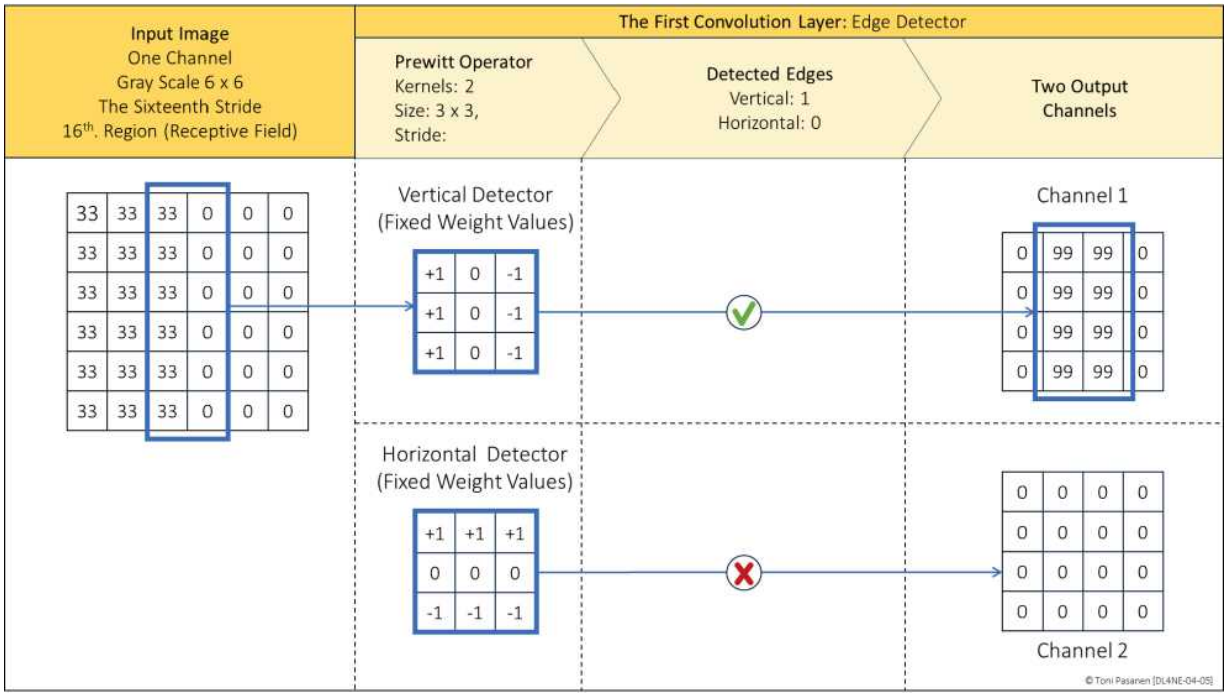


Figure 4-5: *Convolution Layer – Detected Edges*

MaxPooling

MaxPooling is used to reduce the size of the output channels if needed. In our example, where the channel size is relatively small (4 × 4), MaxPooling is unnecessary, but we use it here to demonstrate the process. Similar to convolution, MaxPooling uses a kernel and a stride. However, instead of fixed weights associated with the kernel, MaxPooling selects the highest value from each covered region. This means there is no computation involved in creating the new matrix. MaxPooling can be considered as a layer or part of the convolution layer. Due to its non-computational nature, I see it as part of the convolution layer rather than a separate layer.

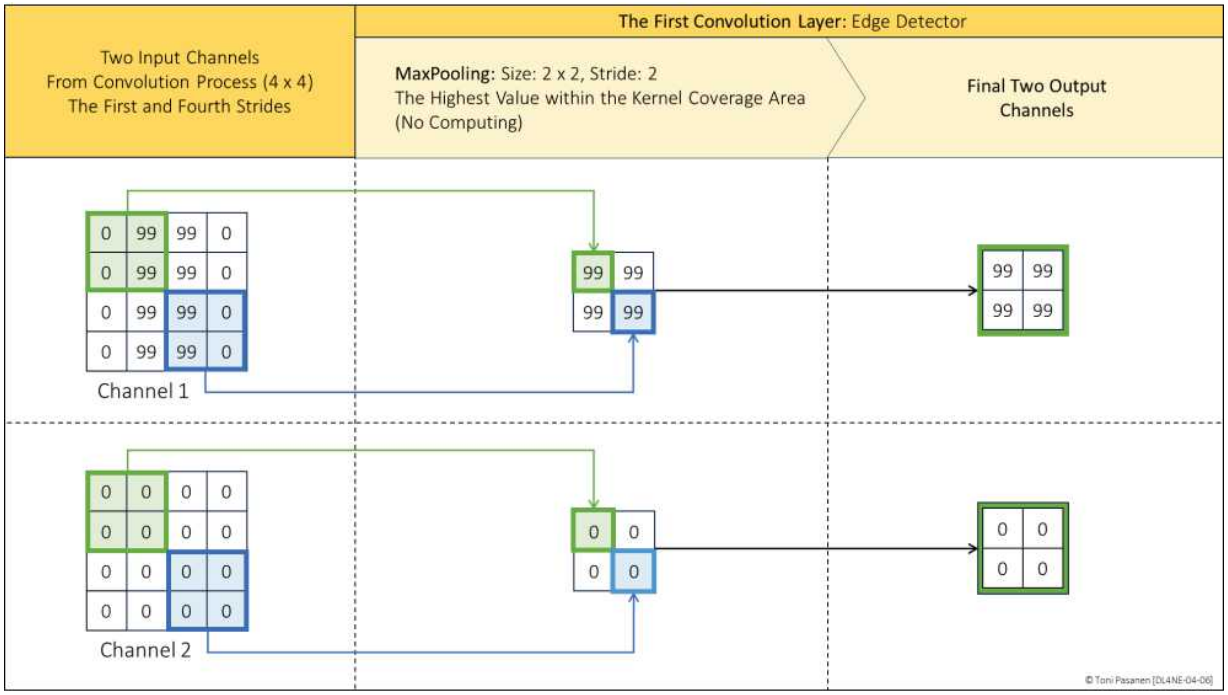


Figure 4-6: *Convolution Layer: MaxPooling*

The First Convolution Layer: Convolution

In this section, we take a slightly different view of convolutional neural networks compared to the preceding sections. In this example, we use the Kirsch operator in the first convolution layer. It uses 8 kernels for detecting vertical, horizontal, and diagonal edges. Similar to the Prewitt operator, the

Kirsch operator uses fixed weight values in its kernels. These values are

shown in Figure 4-7.

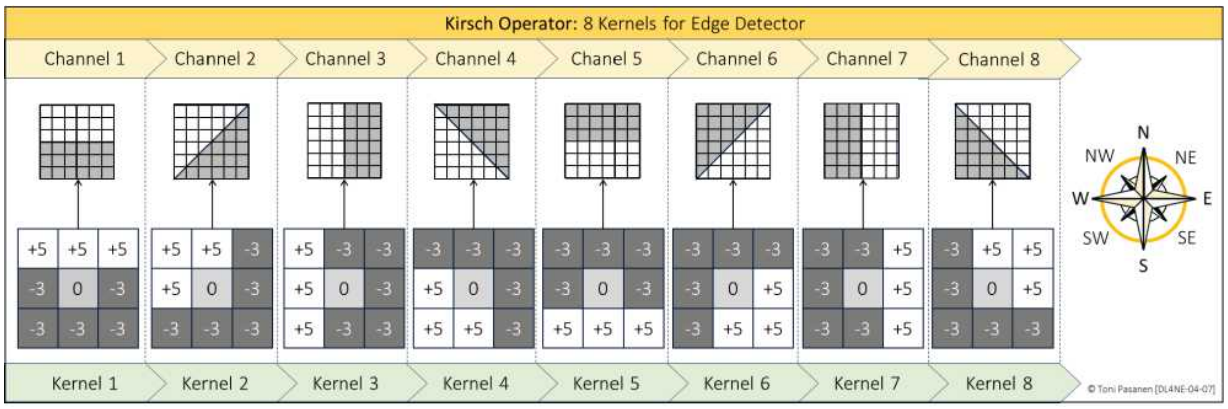


Figure 4-7: *Kirsch Operator.*

In Figure 4-8, we use a pre-labeled 96 x 96 RGB image for training. An RGB image has three color channels: red, green, and blue for each pixel. It is possible to apply all kernels to each color channel individually, resulting in 3 x 8 = 24 output channels. However, we follow the common practice of applying the kernels to all input channels simultaneously, meaning the eight Kirsch kernels have a depth of 3 (matching the RGB channels). Each kernel processes the RGB values together and produces one output channel. Thus, each neuron uses 3 (width) x 3 (height) x 3 (depth) = 27 weight parameters for calculating the weighted sum. With a stride value of one, the convolution process generates eight 94 x 94 output channels. The formula for calculating weighted sum:

3

zz=^

ww=1

33

$^{\wedge}{}^{\wedge}(iiiiiiiiii\,[ww, h, dd]\,xx\,kkiikkkkiikk\,[ww, h, dd]) + yyiiiibb\,h{=}1$ $dd = 1$

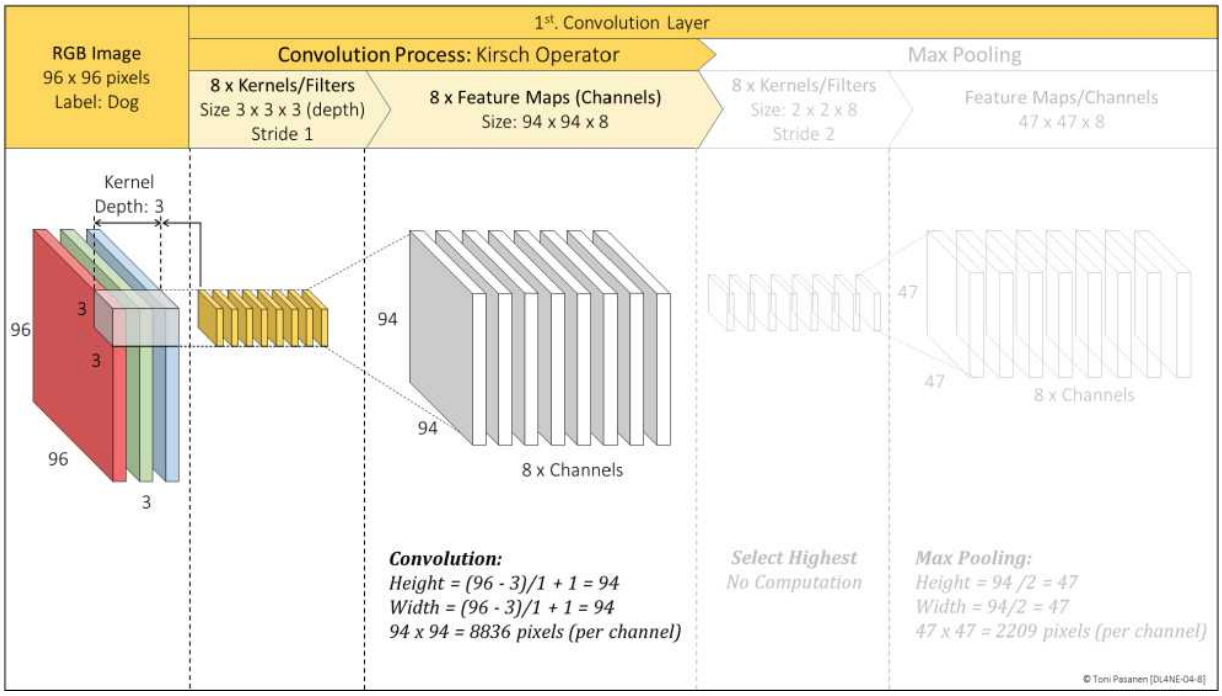


Figure 4-8: *The First Convolution Layer – Convolution Process.*

The First Convolution Layer: MaxPooling

To reduce the size of the output channels from the first convolution layer, we use MaxPooling. We apply eight 2 x 2 kernels, each with a depth of 8, corresponding to the output channels. All kernels process the channels simultaneously, selecting the highest value among the eight channels. MaxPooling with this setting reduces the size of each output channel by half, resulting in eight 47 x 47 output channels, which are then used as input channels for the second convolution layer.

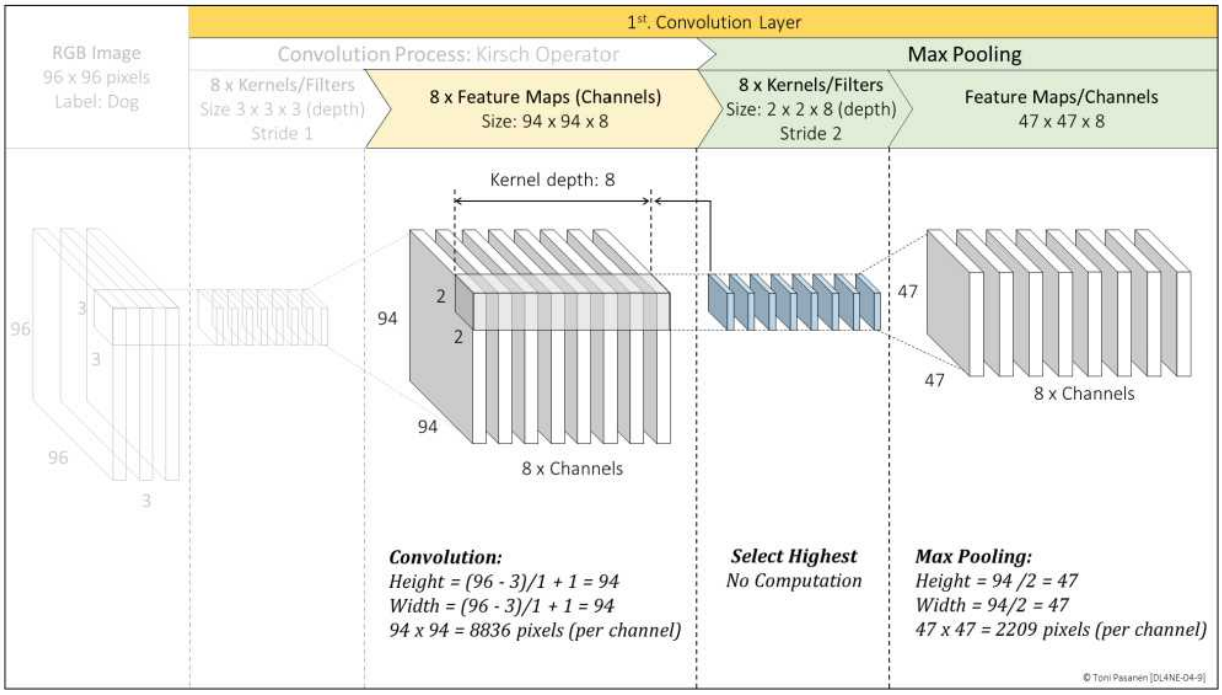


Figure 4-9: *The First Convolution Layer – MaxPooling.*

The Second Convolution Layer

Figure 4-10 shows both the convolution and MaxPooling processes. The eight 47 x 47 output channels produced by the first convolution layer are used as input channels for the second convolution layer. In this layer, we use 16 kernels whose initial weight values are randomly selected and adjusted during the training process. The kernel size is set to 3 x 3, and the depth is 8, corresponding to the number of input channels. Thus, each kernel calculates a weighted sum over 3 x 3 x 8 = 72 parameters with 72 weight values. All 16 kernels produce new 45 x 45 output channels by applying the ReLU activation function. Before flattening the output channels, our model applies a MaxPooling operation, which selects the highest value within the kernel

coverage area (region). This reduces the size of the output channels by half, from 45 x 45 to 22 x 22.

If we had used the original image without convolutional processing as input to the fully connected layer, there would have been 27,648 input parameters (96 x 96 x 3). Thus, the two convolution layers reduce the number of input parameters to 7,744 (22 x 22 x 16), which is approximately a 72% reduction.

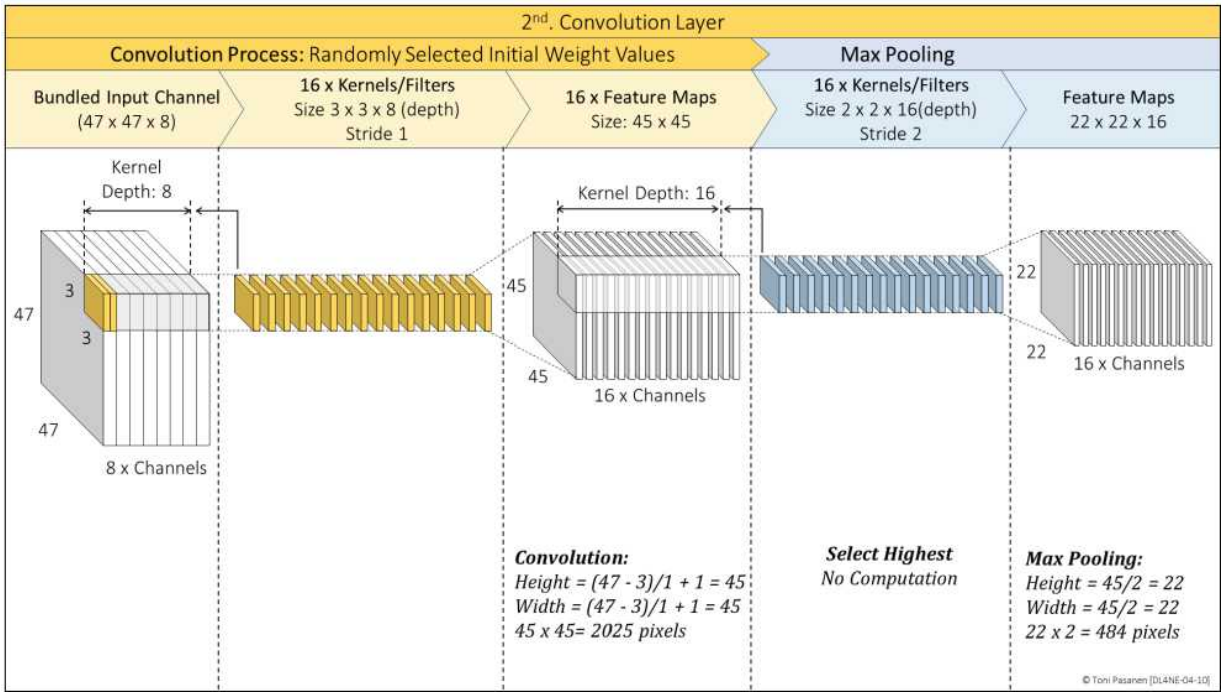


Figure 4-10: *The Second Convolution Layer – Convolution and MaxPooling.*

FULLY CONNECTED LAYERS

Before feeding the data into the fully connected layer, the multidimensional 3D array (3D tensor) is converted into a 1D vector. This produces 7,744 input values (22 x 22 x 16) for the input layer. We use 4,000 neurons with the ReLU activation function in the first hidden layer, which is approximately half the number of input values. In the second hidden layer, we have 1,000 neurons with the ReLU function. The last layer, the output layer, has 10 neurons using the SoftMax function.

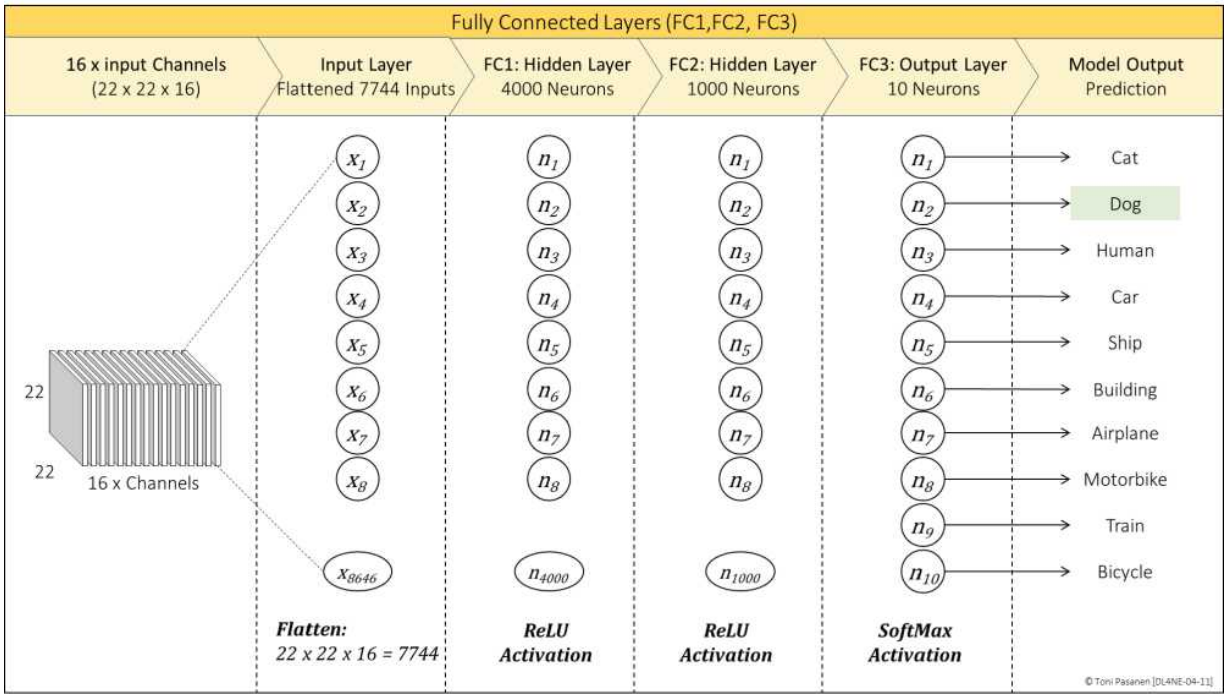


Figure 4-11: *Fully Connected Layer – Convolution and MaxPooling.*

BACKPROPAGATION PROCESS

In Fully Connected Neural Networks (FCNNs), every neuron has its own unique set of weights. In contrast, Convolutional Neural Networks (CNNs) use parameter sharing, where the same filter (kernel) is applied across the entire input image. This approach not only reduces the number of parameters but also enhances efficiency.

Additionally, backpropagation in CNNs preserves the spatial structure [1] of the input data through convolution and pooling operations. This helps the network learn spatial features like edges, textures, and patterns. In contrast, FNNs flatten the input data into a 1D vector, losing any spatial information and making it harder to capture meaningful patterns in images.

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Yann LeCun, Corina Cortes, Christoper J.C. Burges: The MNIST database of handwritten digits. October 2010.

https://www.lri.fr/~marc/Master2/MNIST_doc.pdf

[3] Goodfellow, I., Bengio, Y., & Courville, A. (2016, November 18). Deep Learning. MIT Press.

https://www.deeplearningbook.org/

[4] IBM. (n.d.). What are Convolutional Neural Networks?

https://www.ibm.com/think/topics/convolutional-neural-networks

[5] Wikipedia contributors. (n.d.). Kirsch operator. Wikipedia. https://en.wikipedia.org/wiki/Kirsch_operator

[6] Wikipedia contributors. (n.d.). Prewitt operator. Wikipedia.

https://en.wikipedia.org/wiki/Prewitt_operator

[1]

Spatial features refer to the characteristics of an image that describe the relationship between pixels based on their positions. These features capture the spatial structure of the image, such as edges, corners, textures, shapes, and patterns, which are essential for recognizing objects and understanding the visual content.

CHAPTER 5: RECURRENT NEURAL NETWORKS

INTRODUCTION

So far, this book has introduced two neural network architectures. The first one, the Feed-Forward Neural Network (FNN), works well for simple tasks, such as recognizing handwritten digits in small-sized images. The second one, the Convolutional Neural Network (CNN), is designed for processing larger images. CNNs can identify objects in images even when the location or orientation of the object changes.

This chapter introduces the Recurrent Neural Network (RNN). Unlike FNNs and CNNs, an RNN's inputs include not only the current data but also all the inputs it has processed previously. In other words, an RNN preserves and uses historical data. This is achieved by feeding the output of the previous time step back into the hidden layer along with the current input vector.

Although RNNs can be used for predicting sequential data of variable lengths, such as sales figures or a patient's historical health records, this chapter focuses on how RNNs can perform character-based text autocompletion. The upcoming chapters will explore word-based text prediction.

TEXT DATASETS

For training the RNN model, we typically use text datasets like *IMDB Reviews* or the *Wikipedia Text Corpus*. However, in this chapter, we simplify the process by using a tailored dataset containing only the word "alley". Figure 5-1 illustrates the steps involved.

1. Splitting the text into characters: First, we break the word into its individual letters (e.g., a, l, l, e, y).
2. Index mapping: Each character is assigned an index number, which maps it to a one-hot-encoded vector. For example, the letter “a” is assigned index 0, corresponding to the one-hot vector [1, 0, 0, 0].
3. Sequence creation: Finally, we define the sequence of characters to predict. For example, when the input character is “a” (input vector [1, 0, 0, 0]), the model should output the letter “l” (output vector [0, 0, 1, 0]).

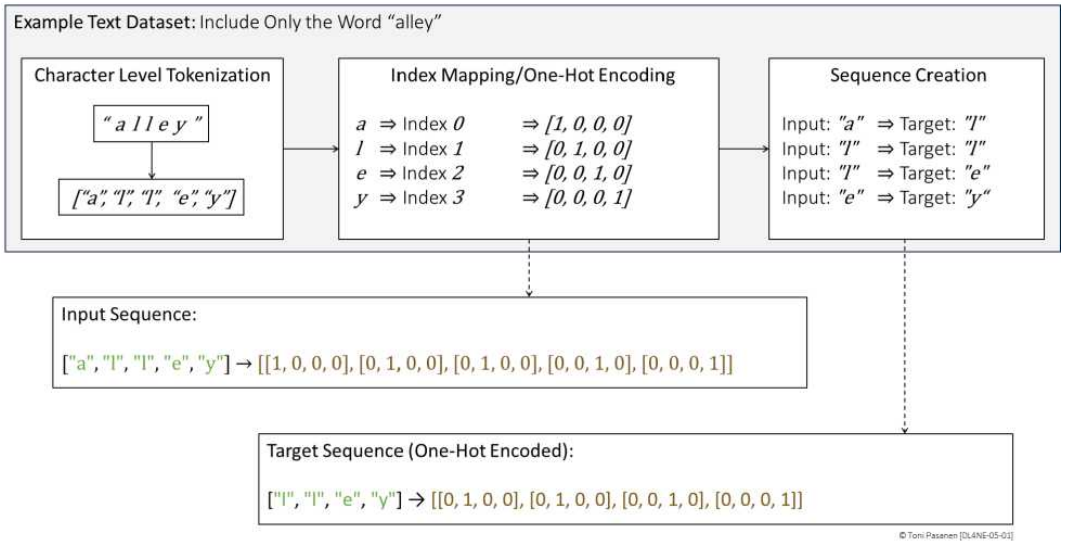


Figure 5-1: *Recurrent Neural Networks – Text Dataset and One-Hot Encoding.*

TRAINING RECURRENT NEURAL NETWORKS

Figure 5-2 illustrates a simplified view of the RNN training process. In the previous section, we explained how one-hot encoding is used to produce an input vector for training. For example, the character “a” is represented by the input vector [1, 0, 0, 0], which is fed into the hidden layer. Each neuron in the hidden layer has its own dedicated weight matrix associated with the input vector.

Weight Matrices in RNNs

The weight values associated with input vectors are denoted as U, while the weights for the recurrent connections (connections between neurons across time steps) are noted as W. This separation is a standard way to distinguish weights for input processing from those used in recurrent operations.

Weighted Sum Calculation in the Hidden Layer

The neurons in the hidden layer calculate the weighted sum of the input vector. Only the sequence corresponding to the 1 in the input vector contributes to the calculation, as all other sequences result in zero when multiplied. This calculation also includes a bias term. For example, if the weight matrix for the

input vector [1, 0, 0, 0] is [$Un1$, $Un2$, $Un3$, $Un4$], only the weight $Un1$ contributes to the sum.

The result of this weighted sum for the initial time step is denoted as $h^{(-1t)}$. This result is "stored" and used as an input for the next time step. After calculating the weighted sum, it is passed through an activation function, and the resulting activation values are fed into the output layer.

Output Layer Operations

In our example, there are two output neurons for simplicity, but in real-life scenarios, the output layer typically contains the same number of neurons as the input vector dimensions (four in this case). Each output neuron calculates a weighted sum of its inputs, producing a value known as a logit. These logits are passed through the SoftMax activation function, which converts them into probabilities for each output neuron. Note, SoftMax function is discussed in chapter 3 – Multi-Class Classification.

In this example, the output neuron with the highest probability corresponds to the third position (not shown in the figure). This results in the output vector [0, 0, 1, 0], which represents the character “l.”

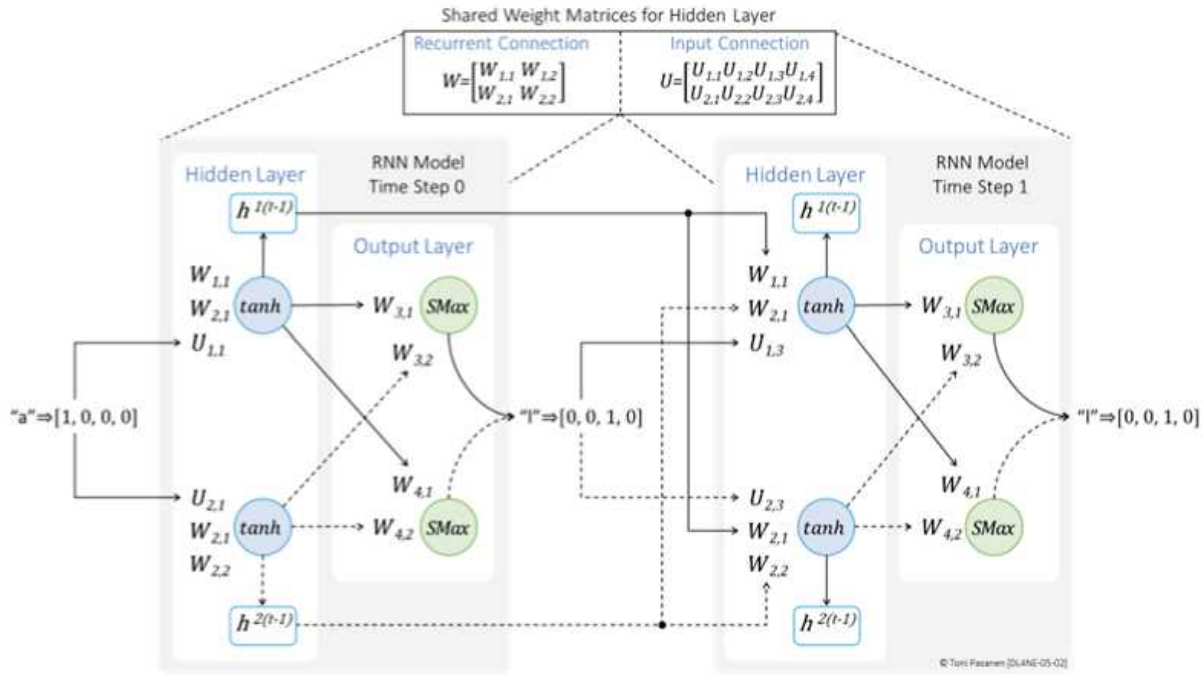


Figure 5-2: *Recurrent Neural Networks – Basic Operation.*

COMPARISON WITH FEED-FORWARD NEURAL NETWORKS (FNNS)

So far, this process resembles that of a Feed-Forward Neural Network (FNN). Input vectors are passed from the input layer to the hidden layer, where the neurons compute weighted sums and apply an activation function. Since the hidden and output layers are fully connected, the hidden layer's activation values are passed to the output layer.

MOVING TO THE SECOND TIME STEP

At the second time step, the output vector [0, 0, 1, 0], along with the weighted sum $h^{n(t-1)}$ from the previous step, is used to calculate the new weighted sum. This calculation also includes a bias term. Since the same model is used at every time step, the weight matrices remain unchanged. At this time step, only the weight $Un3$ contributes to the sum, as it corresponds to the non-zero value in the input vector. The rest of the process follows the same steps as in the initial time step. Once time step 1 is completed, the process advances to time step 2, repeating the same calculations. This sequence continues until the training is completed.

BACKWARD PASS IN RECURRENT NEURAL NETWORKS

The backward pass in RNNs is called Backpropagation Through Time (BPTT) because it involves propagating errors not only through the network layers but also backward through time steps. If you think of time steps as stacked layers, the BPTT process requires fewer computation cycles and memory than Feed-Forward Neural Network (FNN), because RNN uses shared weight matrices across the layers while FMM has assigned per-layer weight values. Like RNN, the Convolutional Neural Network (CNN), introduced in Chapter 4, leverages shared weight matrices but within a layer not between the layers.

CHALLENGES OF A RNN MODELL

Figure 5-3 shows the last two time steps of our Recurrent Neural Network (RNN). At the time step *n* (on the left side), there are two inputs for the weighted sum calculation: X*n* (the input at the current time step) and *ht*-1 (the hidden state from the previous time step).

First, the model calculates the weighted sum of these inputs. The result is then passed through the neuron's activation function (Sigmoid in this example). The output of the activation function, *ht* , is fed back into the recurrent layer on the next time step, *n*+1. At time step *n*+1, the *ht* is combined with the input X*n* to calculate weighted sum. This result is then passed through the activation function, which now produces the model's prediction, *yy*^ (y hat). These steps are part of the Forward Pass process.

As the final step in the forward pass, we calculate the model's accuracy using the Mean Square Error (MSE) function (explained in Chapter 2).

If the model's accuracy is not close enough to the expected result, it begins the *Backward Pass* to improve its performance. The most used optimization algorithm for minimizing the loss function during the backward pass is *Gradient Descent*, which updates the model's parameters step by step.

The backward pass process starts by calculating the derivative of the error function (i.e., the gradient of the error function with respect to the output activation value) to determine the Output Error.

Next, the Output Error is multiplied with the derivative of the activation function to compute the local Error Term for the neuron. (i.e., the derivative of the activation function with respect to its input is the local gradient, which determines how the activation values changes in response to its input change.) The error terms are then propagated through all time steps to calculate the actual Weight Adjustment Values.

In this example, we focus on how the weight value associated with the recurrent connection is updated. However, this process also applies to

weights linked to the input values. The neuron-specific weight adjustment values are calculated by multiplying the local error term with the corresponding input value and the learning rate.

The difference between the backward pass process in a Feedforward Neural Network (FNN) and a Recurrent Neural Network (RNN) is that the RNN uses *Backpropagation Through Time* (BPTT). In this method, the weight adjustment values from each time step are accumulated during backpropagation. Optionally, these accumulated gradients can be averaged over the number of time steps to prevent the gradient magnitude from becoming too large for long sequences. This averaging is the default behavior in implementations using TensorFlow and

PyTorch frameworks (PyTorch is explained in detail in Chapter 14).

Since the RNN model uses shared weight matrices across all time steps, only one weight parameter per recurrent connection needs to be updated. In this simplified example, we have one recurrent connection because there is only one neuron in the recurrent layer. However, in real-world scenarios, RNN layers often have hundreds of neurons and thousands of time steps.

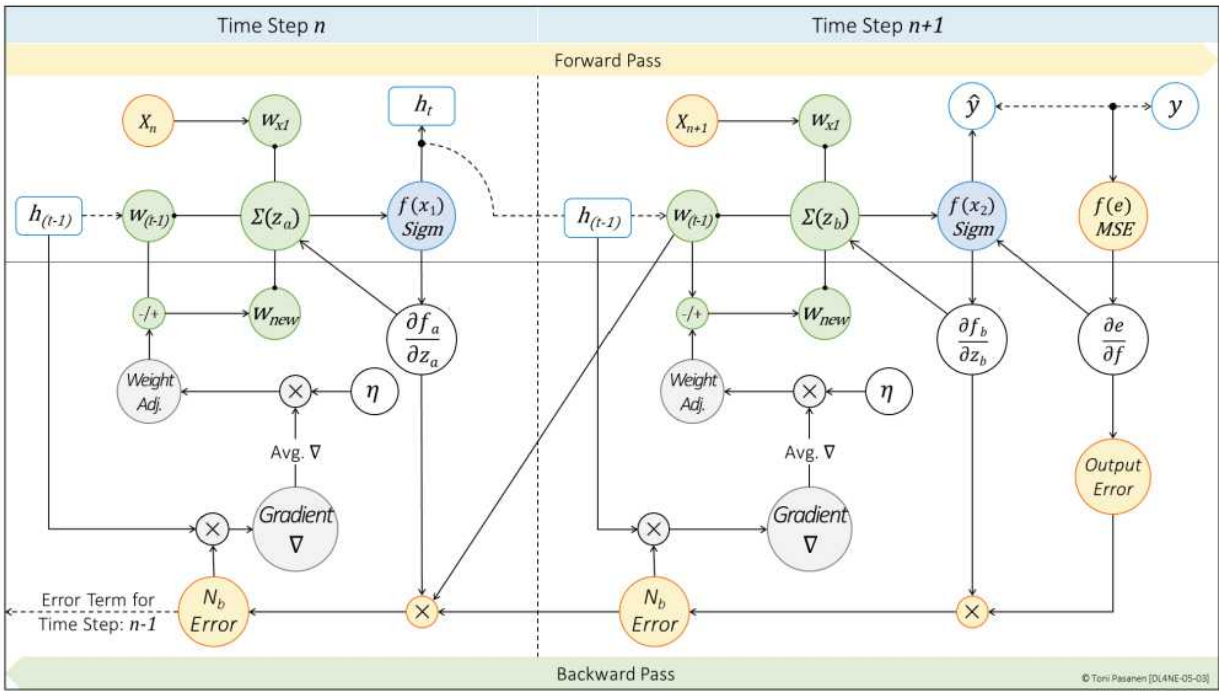


Figure 5-3: *Overview of the Weight Adjustment Process.*

Saturated Neurons

Figure 5-4 depicts the S-curve of the Sigmoid activation function. It shows how the output of the function (y) changes in response to variations in the input (z). The chart illustrates how the rate of change slows down significantly when the input value

exceeds 2.2 or falls below -2.2. Beyond these thresholds, approaching input values of 5.5 and -5.5, the rate of change becomes negligible from a learning perspective. This behavior can occur due to a poor initial weight assignment strategy, where the initial weight values are either too small or too large, potentially causing backpropagation through time (BPTT) to adjust the weights in the wrong direction. This issue is commonly known as *neuron saturation*.

Another issue illustrated in the figure is that the Sigmoid activation function output (y) is practically zero when the input value is less than -5. For example, with z = -5, y = 0.0998, but with z = -7, y drops to just 0.0009. The problem with these "almost-zero" output values is that the neuron becomes "dead," meaning its output (y) has negligible impact on the model's learning process. In an RNN model, where the neuron's output is reused in the recurrent layer as the hidden state (h), a close-to-zero value causes the neuron to "forget" inputs from preceding time steps.

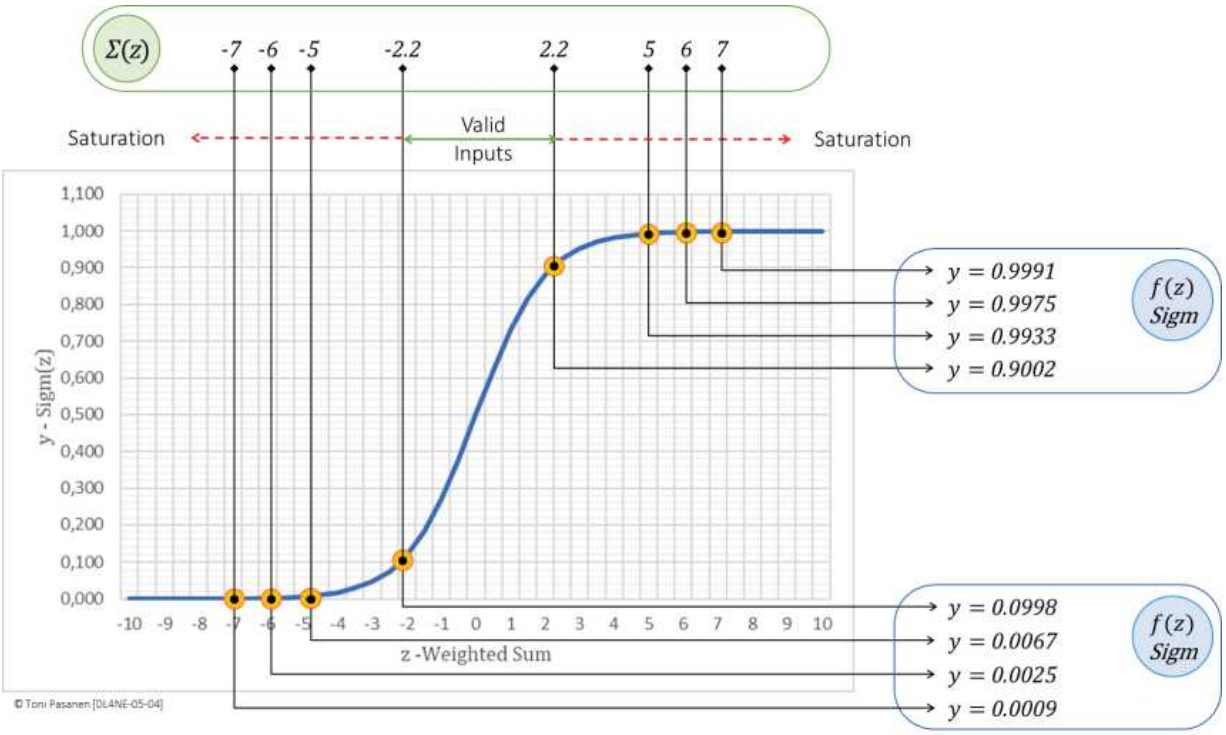


Figure 5-4: *The Problem with the S-curved Sigmoid Function*

Figure 5-5 illustrates a hypothetical RNN with five time steps. This example demonstrates how some recurrent connections for the hidden state values (h) can become insignificant from the perspective of subsequent time steps. For instance, if the output of the Sigmoid activation function at time step 1 (h1) is 0.0007, the corresponding value at time step 2 (h2) would increase by only 0.0008 compared to the scenario where h1 is zero.

Similarly, in an RNN with 1000 time steps, the learning process is prone to the vanishing gradient problem during backpropagation through time (BPTT). As gradients are propagated backward across many time steps, they often shrink exponentially due to repeated multiplication by small values (e.g., derivatives of activation Sigmoid functions). This can cause the learning curve to plateau or decrease, leading to poor weight

updates and suboptimal learning. In severe cases, the learning process may effectively stop, preventing the model from achieving the expected performance.

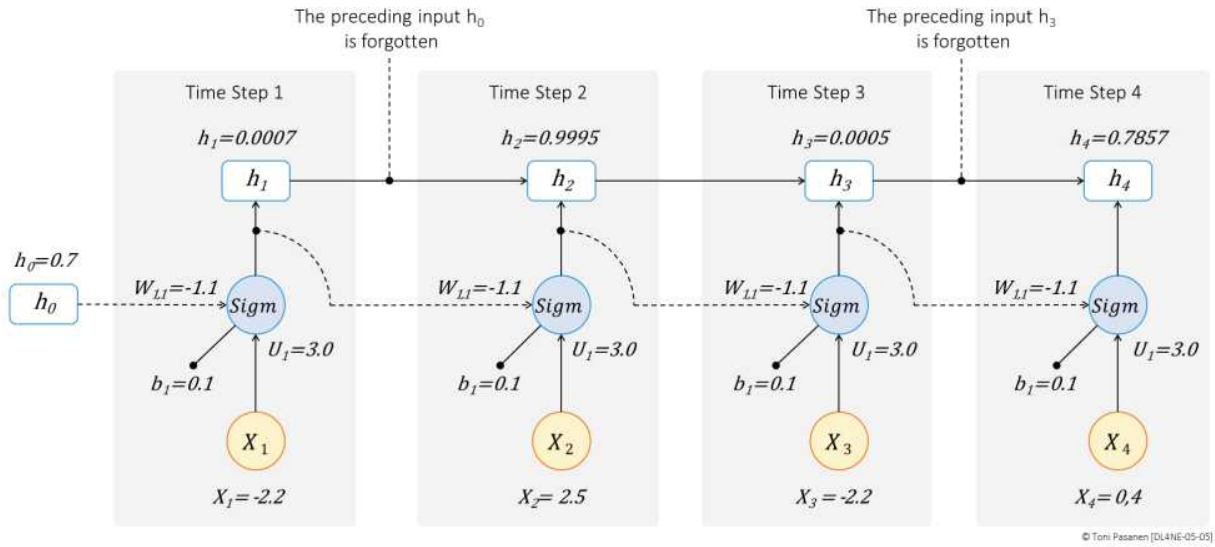


Figure 5-5: *RNN and "Forgotten" History.*

When using a data parallelization strategy with Recurrent Neural Networks (RNNs), input data batches are distributed across multiple GPUs, each running the same model independently on its assigned batch. During the backpropagation through time (BPTT) process, each GPU calculates gradients locally for its portion of the data. These gradients are then synchronized across all GPUs, typically by averaging them, to ensure consistent updates to the shared model parameters.

Since the weight matrices are part of the shared model, the updated weights remain synchronized across all GPUs after each training step. This synchronization ensures that all GPUs use the same model for subsequent forward and backward passes.

However, due to the sequential nature of RNNs, BPTT must compute gradients step by step, which can still limit scalability when dealing with long sequences. Despite this, data parallelization accelerates training by distributing the workload and reducing the computational burden for each GPU.

We can also implement the model parallelization strategy with RNNs, which synchronizes both activation values during the forward pass and gradients during backpropagation.

The parallelization strategy significantly affects network utilization due to the synchronization process, specifically, what we synchronize and at what rate. Several upcoming chapters will focus on different parallelization strategies.

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016, November 18). Deep Learning. MIT Press.

https://www.deeplearningbook.org/

[3] Schmidt, R. M. (2019, November 23). Recurrent Neural Networks (RNNs): A Gentle Introduction and Overview. arXiv.

https://arxiv.org/abs/1912.05911

[4] Wikipedia contributors. (n.d.). Recurrent Neural Network. Wikipedia.

https://en.wikipedia.org/wiki/Recurrent_neural_network

[5] Britz, D. (2015, September 17). Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs. Denny's Blog.

https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-1/

[6] Amidi, A., & Amidi, S. (n.d.). CS 230 - Recurrent Neural Networks Cheatsheet. Stanford University.

https://web.stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

[7] Brownlee, J. (2022, August 15). An Introduction to Recurrent Neural

Networks and the Math That Powers Them. Machine Learning Mastery.

https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/

CHAPTER 6: LONG SHORT-TERM MEMORY

INTRODUCTION

As mentioned in the previous chapter, Recurrent Neural Networks (RNNs) can have hundreds or even thousands of time steps. These basic RNNs often suffer from the gradient vanishing problem, where the network struggles to retain historical information across all time steps. In other words, the network gradually "forgets" historical information as it progresses through the time steps.

One solution to address the horizontal gradient vanishing problem between time steps is the use of Long Short-Term Memory (LSTM) based RNN instead of basic RNN. LSTM cells can preserve historical information across all time steps, whether the model contains ten or several thousand time steps.

Figure 6-1 illustrates the overall architecture of an LSTM cell. It includes three gates: the *Forget gate*, the *Input gate* (a.k.a. *Remember gate*), and the Output gate. Each gate contains input neurons that use the Sigmoid activation function. The reason for employing the Sigmoid function, as shown in Figure 5-4 of the previous chapter, is its ability to produce outputs in the range of 0 to 1. An output of 0 indicates that the gate is "closed," meaning the information is excluded from contributing to the cell's internal state calculations. An output of 1, on the other hand, means that the information is fully utilized in the computation. However, the sigmoid function never gives an exact output of

zero. Instead, as the input value becomes more and more negative (approaching negative infinity), the output gets closer and closer to zero, but it never actually reaches it. Similarly, the sigmoid function's output approaches one as the input value becomes very large (approaching positive infinity). However, just like with zero, the function never exactly reaches one; it only gets very close.

As a one way of completely closing any of the gates, you may set a threshold value manually and define, for example, that the outputs less than 0.01 are interpreted as zero (gate closed). The same principle applies to gate opening, you can set the threshold to, for example, output higher than 0.95 are interpreted as one (gate fully open). However, instead of hard coded threshold, consider alternatives like smooth activation adjustments.

This gating mechanism enables LSTM cells to selectively retain or discard information, allowing the network to manage long-term dependencies effectively.

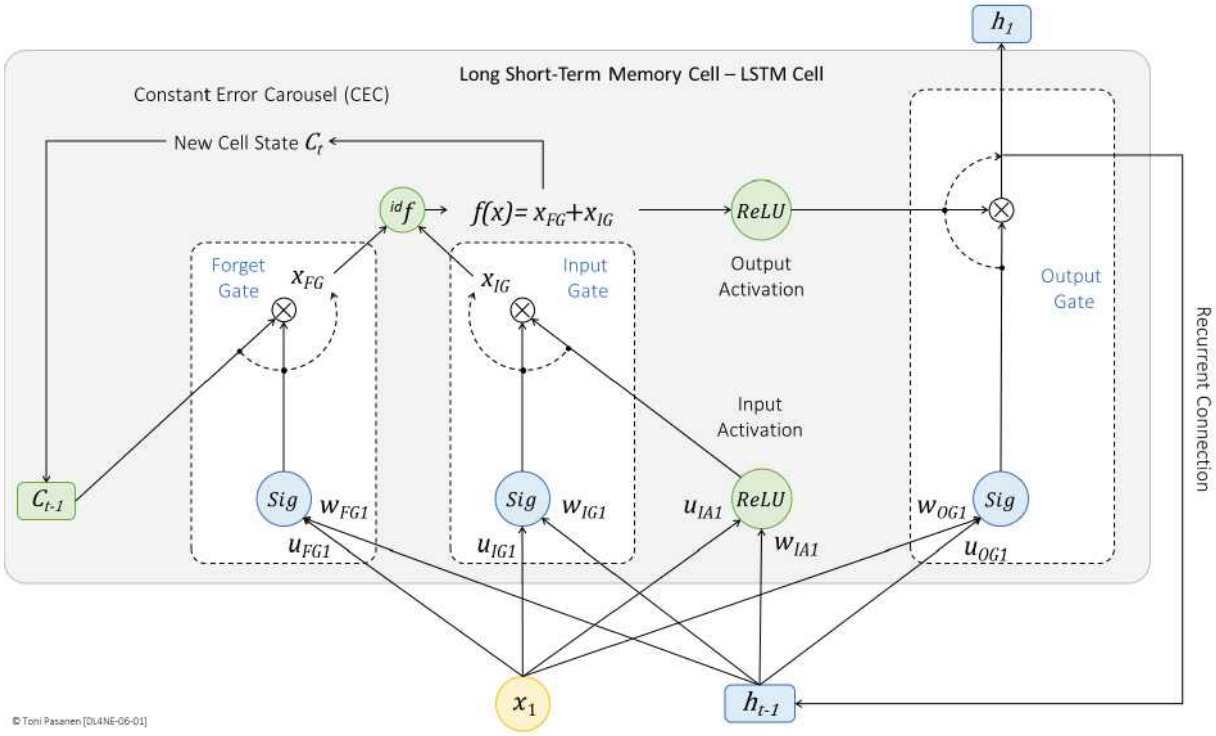


Figure 6-1: *Long Short-Term Memory Cell – Architectural Overview.*

LTSM CELL OPERATION

In addition to producing input for the next layer (if one exists), the output (*h*) of the LSTM cell serves as input for the next time step via the recurrent connection. This process is like how neurons in a basic RNN operate. The LSTM cell also has a *cell state* (C), which is used to retain historical information utilizes the *Constant Error Carousel* (CEC) mechanism, which feeds back the cell state (C) into the computation process where the new cell state is calculated. The following sections briefly describe the processes how an LSTM cell computes the cell state (C), and the cell output (h), and explains the role of the gates in the process.

Forget Gate

The Forget Gate (FG) adjusts the extent to which historical data is preserved. In Figure 6-2, the cell state Ct-1 represents historical data computed by the identity function during a preceding time step. The cell state (C) represents an LSTM cell internal state, not the LSTM cell output (*h*), and it is used for protecting historical data for gradient vanishing during the BPTT. The adjustment factor for Ct-1 is calculated by a neuron using the Sigmoid activation function within the FG.

The neuron in the FG uses shared, input specific weight matrices for the input data (X1) and the input received from the

preceding LSTM cell's output (h_{t-1}). These weight matrices are shared across FG neurons over all time steps, like the approach used in a basic Recurrent Neural Network (RNN). As described in the previous chapter, this sharing reduces the computational requirements for calculating weight adjustment values during Backpropagation Through Time (BPTT). Additionally, the shared weight matrices help reduce the model memory utilization by limiting the number of weight variables.

In the figure, the matrix W_{FG1} is associated with the input received from the preceding time step, while the matrix U_{FG1} is used for the new input value X_1. The weighted sum $(W_{FG1} \cdot h_{t-1}) + (U_{FG1} \cdot X_1)$ is passed through the Sigmoid activation function, which produces the adjustment factor for the cell state value C_{t-1}. The closer the output of the sigmoid function is to the value one, the more the original value affects the calculation of the new value. The same applies to opposite direction, the closer the output of the sigmoid function is to zero, the less the original value affects the calculation of the new value.

Finally, the output of the FG, referred to as X_{FG}, is computed by multiplying the Sigmoid output by the cell state C_{t-1}.

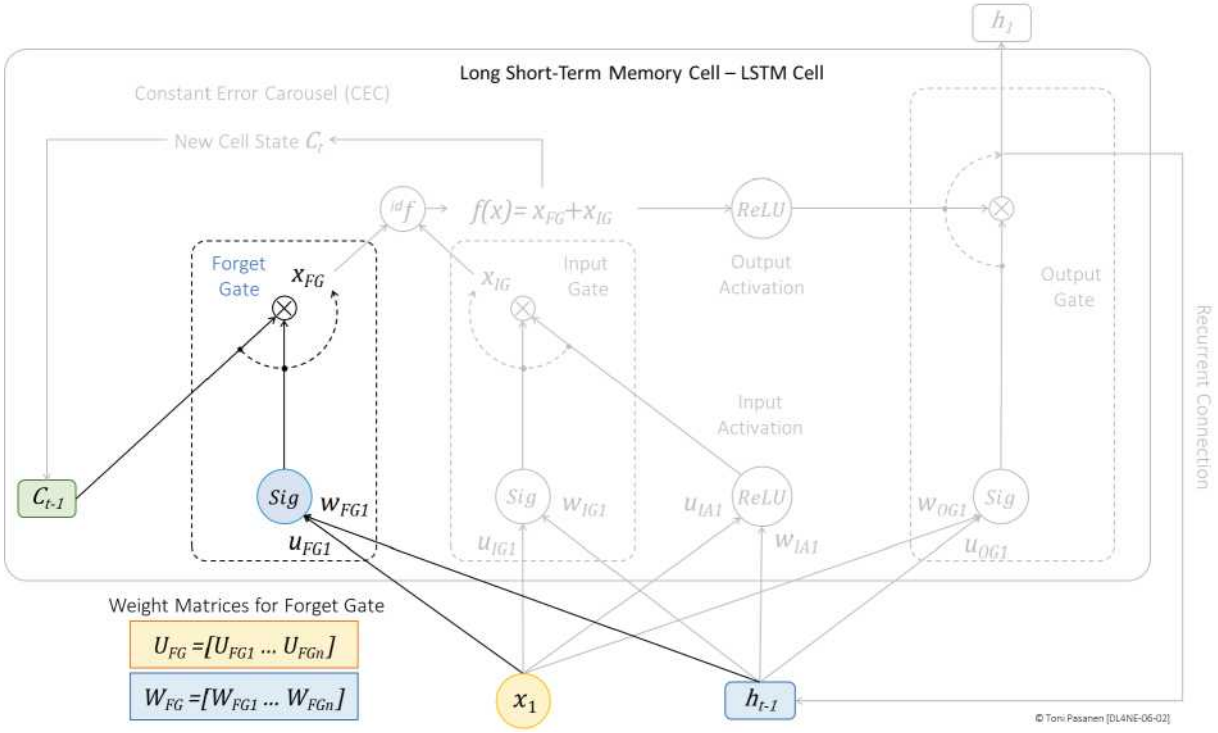


Figure 6-2: *Long Short-Term Memory Cell – Forget Gate.*

Input Gate

The *Input Gate* (IG) determines to what extent the input X1 and the output *ht*-1 from the preceding time step affect the new cell state Ct. For this process, the LSTM cell has two neurons. In Figure 6-3, the internal neuron of IG uses the Sigmoid function, while the *Input Activation* neuron leverages the ReLU function. Both neurons use input-specific weight matrices in the same way as the Forget Gate. The *Input Gate* neuron feeds the weighted sum (*WIG*1 · *ht*-1) + (*UIG*1 · X1) to the sigmoid function. The output determines the proportion in which new input values X1 and *ht*-1 influence the computation of the cell's internal value. The closer the sigmoid function's output is to one, the more the original value influences the new value. Conversely, the closer the output

is to zero, the less it influences the new value. The *Input Activation* neuron feeds the weighted sum (WIA1 · ht-1) + (UIA1 · X1) to the ReLU function. The output is then multiplied by the output of the Sigmoid function, providing the result of the Input Gate. At this phase, the LSTM cell has computed output for both Forget Gate (XFG) and Input Gate (XIG). Next, the LSTM feeds these values to the *Identification Function*.

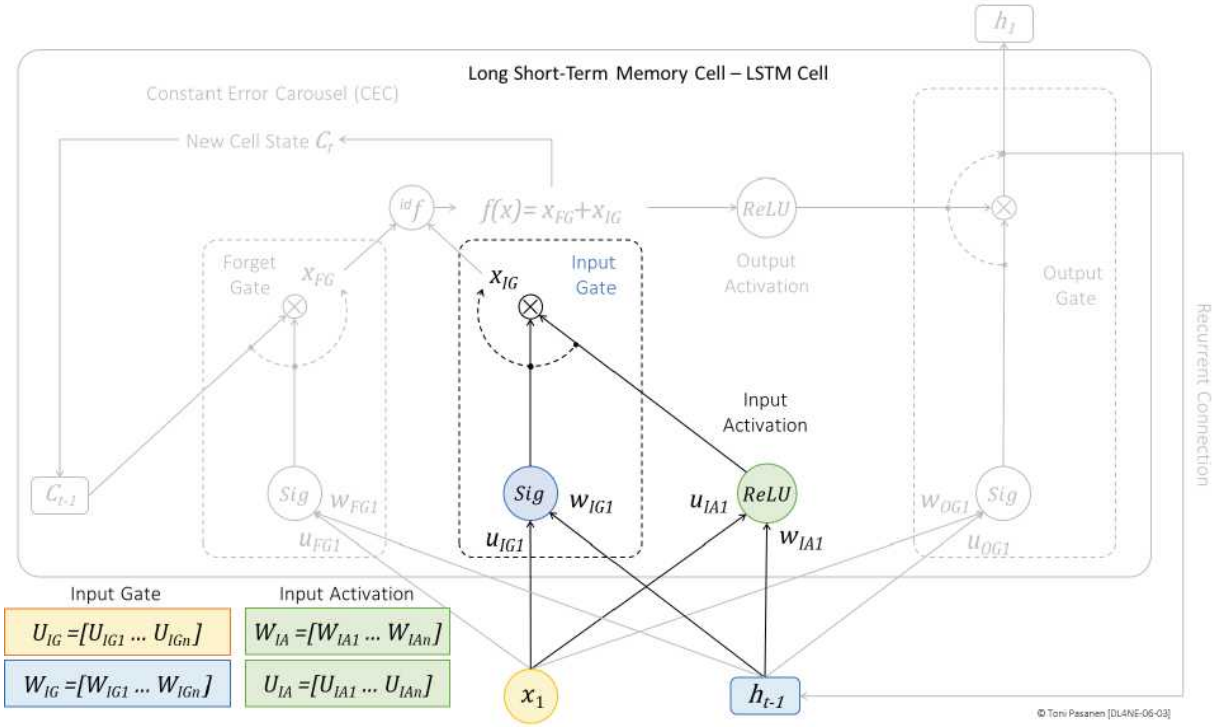


Figure 6-3: *Long Short-Term Memory Cell – Input Gate.*

Output Gate

The *Output Gate* determines whether the output of the Output Activation neuron (ReLU) is fully published, partially published, or left unpublished. The factor of the Output Gate is calculated based on the input value X1 and the output ht-1 from the

previous time step. That said, all Sigmoid neurons and the ReLU Input Activation function use the same inputs, and they leverage shared weight matrices. The input to the *Output Activation* neuron is the sum of the outputs from the Forget Gate (XFG) and the Input Gate (XIG). In the figure, the sum is represented as f(x) = XFG + XIG. The operation is computed by a neuron that uses the Identification function (IDF). The original output of the Identification function is preserved as the internal cell state (C) for the next time step through the CEC (Constant Error Carousel) connection. The output of the Identification Function is then passed to the ReLU *Output Activation* function. This output is multiplied by the result of the Output Gate, producing the actual cell output h1. This value serves as input to the same LSTM cell in the next time step. In a multilayer model, the cell output is also used as input for the subsequent layer.

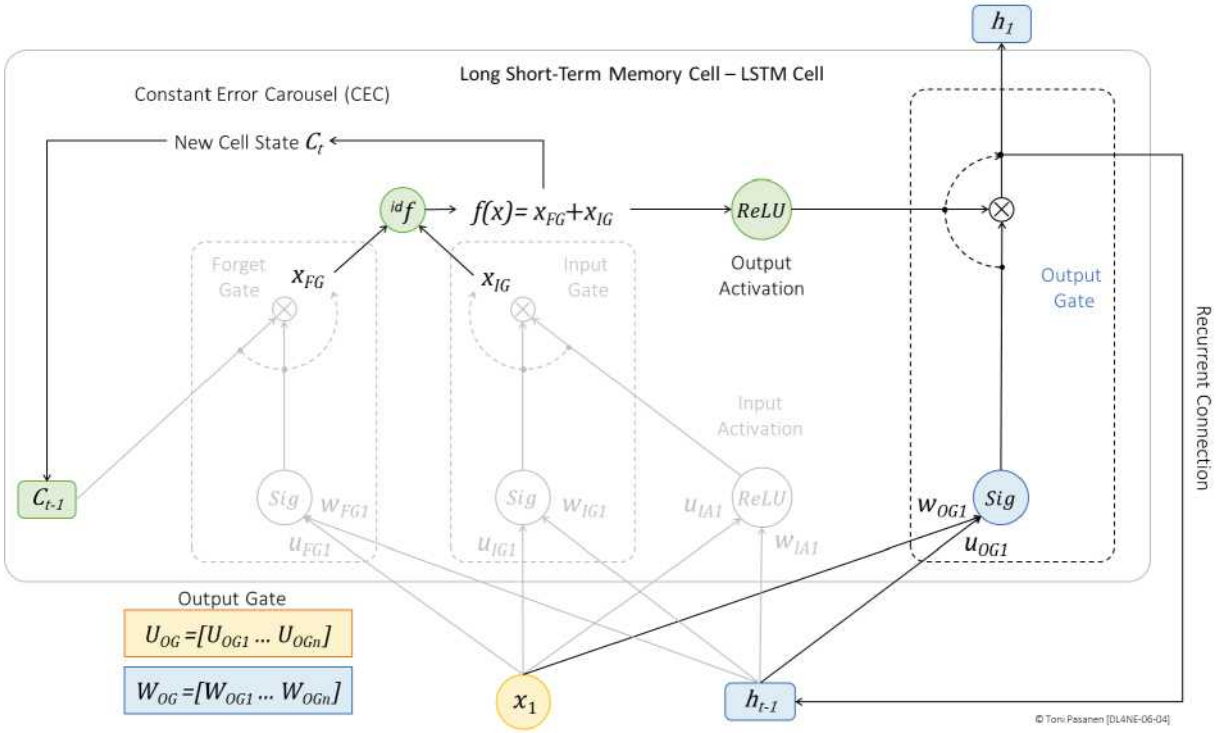


Figure 6-4: *Long Short-Term Memory Cell – Output Gate*.

LTSM-BASED RECURRENT NEURAL NETWORK

Recap of the Operation of an LSTM Cell

The previous section introduced the construction and operation of a single Long Short-Term Memory (LSTM) cell. This section briefly discusses an LSTM-based Recurrent Neural Network (RNN). Before diving into the details, let's recap how an individual LSTM cell operates with a theoretical, non-mathematical example.

Suppose we want our model to produce the sentence: "*It was cloudy, but it is raining now*." The first part of it refers to the past, and one of the LSTM cells has stored the tense "*was*" in its internal cell state. However, the last portion of the sentence refers to the present. Naturally, we want the model to forget the previous tense "*was*" and update its state to reflect the current tense "*is*."

The *Forget Gate* plays a role in discarding unnecessary information. In this case, the forget gate suppresses the word "*was*" by closing its gate (outputting 0). The *Input Gate* is responsible for providing a new candidate cell state, which in this example is the word "*is*." The input gate is fully open (outputting 1) to allow the latest information to be introduced.

The Identification function computes the updated cell state by summing the contributions of the forget gate and the input gate. This updated cell state represents the memory for the next time step. Additionally, the updated cell state is passed through an *Output Activation* function, which provides the cell's output.

The *Output Gate* controls how much of this activated output is shared as the public output. In this example, the output gate is fully open (outputting 1), allowing the word "is" to be published as the final output.

AN OVERVIEW OF AN LSTM-BASED RNN

Figure 6-5 illustrates an LSTM-based RNN model featuring two LSTM layers and a SoftMax layer. The input vectors X1 and X2, along with the cell output ht-1 from the previous time step, are fed into all LSTM cells in the input layer. To keep the figure simple, only two LSTM cells are shown per layer.

The input vectors pass through gates, producing both the internal cell state and the cell output. The internal states are stored using a Constant Error Carousel (CEC) to be utilized in subsequent time steps. The cell output is looped back as an input vector for the next time step. Additionally, the cell output is passed to all LSTM cells in the next layer.

Finally, the SoftMax layer generates the model's output. Note that Figure 6-5 depicts a single time step.

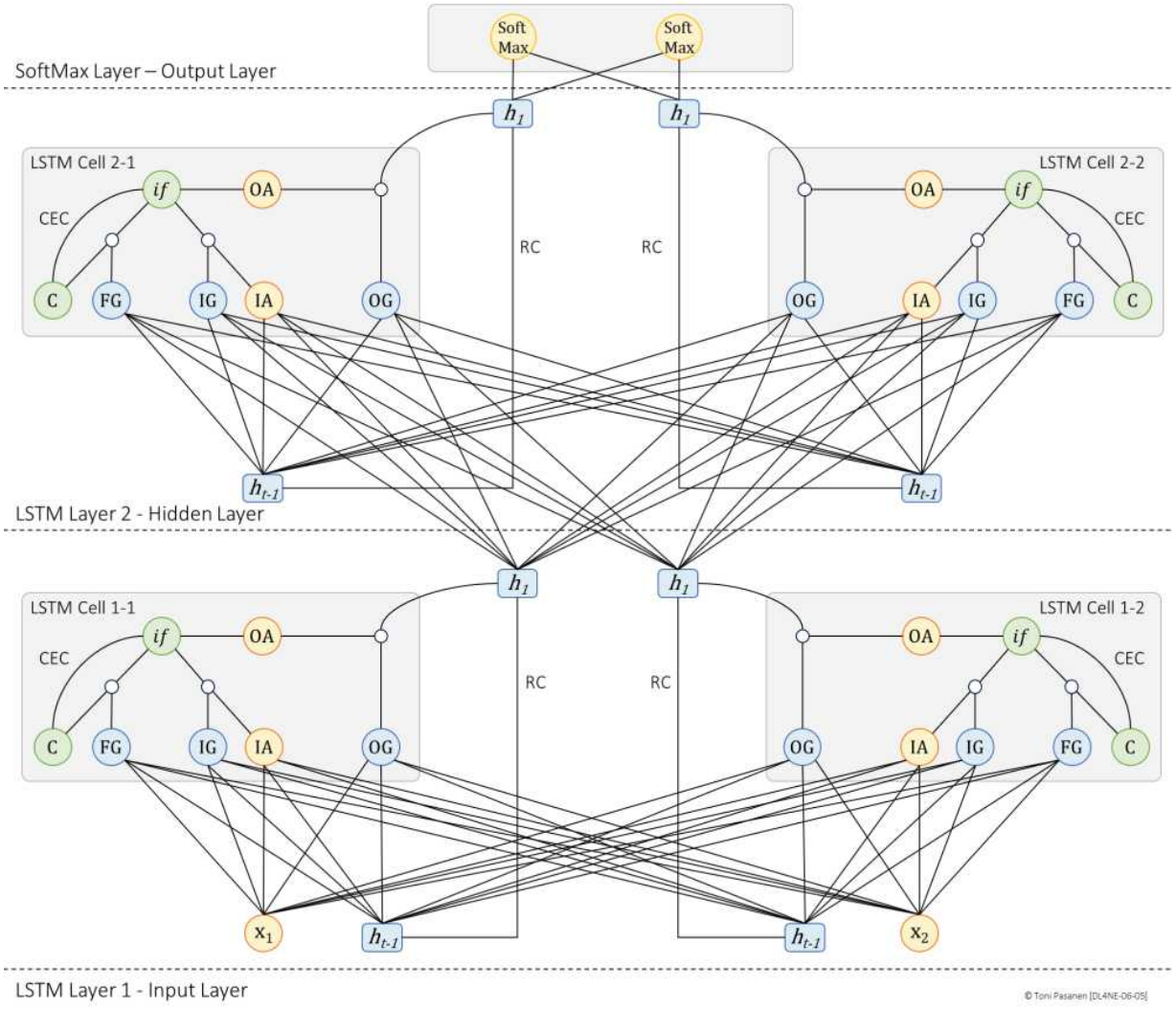


Figure 6-5: *LSTM based RNN Layer Model.*

Figure 6-6 illustrates a layered LSTM-based Recurrent Neural Network (RNN) model that processes sequential data across four time steps. The model consists of three layers: the input LSTM layer, a hidden LSTM layer, and a SoftMax output layer. Each gray square labeled "LSTM" represents a layer containing n LSTM cells.

At the first time step, the input value x1 is fed to the LSTM cells in the input layer. Each LSTM cell computes its internal cell state

(C), applies it to the output activation function, and produces a cell output (ht). This output is passed both to the LSTM cells in the next time step via recurrent connections and to the LSTM cells in the hidden layer at the same time step as an input vector.

The LSTM cells in the hidden layer repeat the process performed by the input layer LSTM cells. Their output (ht) is passed to the SoftMax layer, which computes probabilities for each possible output class, generating the model's predictions (y1). The cell output is also passed to the next time step on the same layer.

The figure also depicts the *autoregressive* mode, where the output of the SoftMax layer at the initial time step t1 is fed back as part of the input for the next time step (t+1) in the input layer. This feedback loop enables the model to use its predictions from previous time steps to inform its processing of subsequent time steps. Autoregressive models are particularly useful in tasks such as sequence generation, where the output sequence depends on previously generated elements.

Key Features Depicted in Figure 6-6

- Recurrent Data Flow: The outputs from each time step are recurrently fed into the next time step, capturing temporal dependencies.
- Layered Structure: The vertical connections between layers allow the model to hierarchically process input data, with higher layers learning progressively abstract features.

- **Autoregressive Feedback:** The use of SoftMax outputs as part of the next time step's input highlights the autoregressive nature of the model, commonly used in sequence prediction and generation tasks.

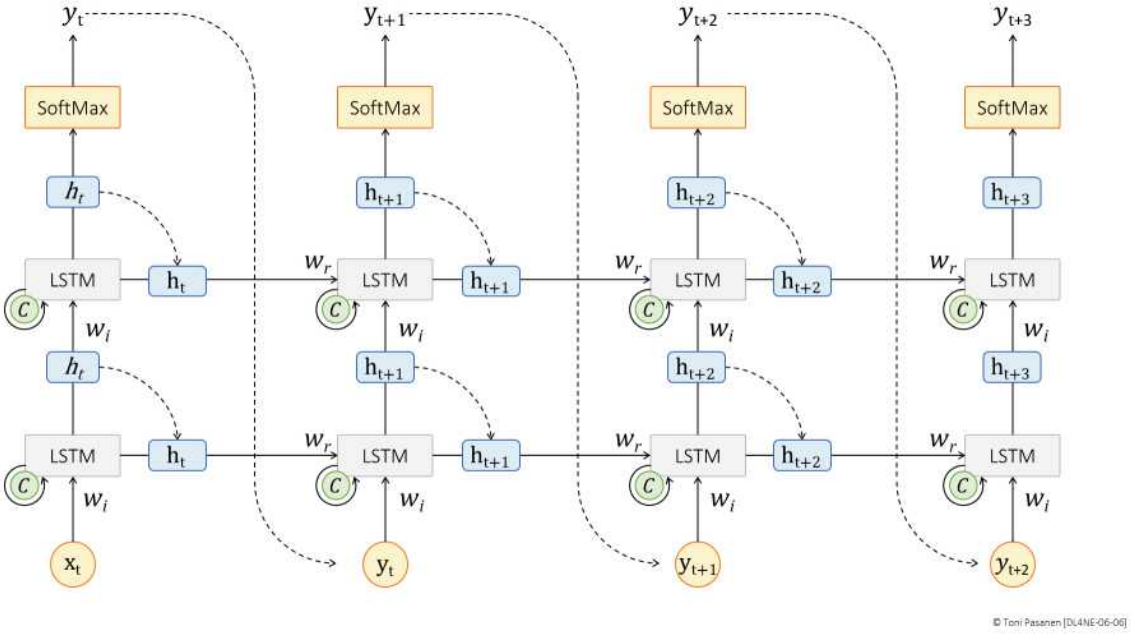


Figure 6-6: *LSTM-Based RNN Model with Layered Structure and Four Time Steps.*

CONCLUSION

Figure 6-6 demonstrates the interplay between sequential and layered data flow in a multi-layered LSTM model, showcasing how information is processed both temporally (across time steps) and hierarchically (across layers). The autoregressive feedback loop further illustrates the model's capability to adapt its predictions based on prior outputs, making it well-suited for tasks such as time series forecasting, natural language processing, and sequence generation.

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Hochreiter, S., & Schmidhuber, J. (1997, November 15). Long Short

Term Memory. Neural Computation, 9(8), 1735–1780.

https://www.bioinf.jku.at/publications/older/2604.pdf

[3] Staudemeyer, R. C., & Rothstein Morris, E. (2019, September 12). Understanding LSTM -- a tutorial into Long Short-Term Memory Recurrent Neural Networks. arXiv preprint arXiv:1909.09586. https://arxiv.org/abs/1909.09586

[4] Colah's Blog. (2015, August 27). Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[5] Dive into Deep Learning. (2020). 10.1. Long Short-Term Memory (LSTM).

https://d2l.ai/chapter_recurrent-modern/lstm.html

[6] Analytics Vidhya. (2022, March 10). An Overview on Long Short Term Memory (LSTM).

https://www.analyticsvidhya.com/blog/2022/03/an-overview-on-long-short-term-memory-lstm/

CHAPTER 7: LARGE LANGUAGE MODEL (LLM)

INTRODUCTION

This chapter introduces the basic operations of Transformer-based Large Language Models (LLMs), focusing on fundamental concepts rather than any specific models, such as OpenAI's GPT (Generative Pretrained Transformer).

The chapter begins with an introduction to tokenization and word embeddings, which convert input words into a format the model can process. Next, it explains how the transformer component leverages decoder architecture for input processing and prediction.

This chapter has two main goals. First, it explains how an LLM understands the context of a word. For example, the word "clear" can be used as a verb (Please, clear the table.) or as an adjective (The sky was clear.), depending on the context. Second, it discusses why LLMs require parallelization across hundreds or even thousands of GPUs due to the large model size, massive datasets, and the computational complexity involved.

TOKENIZER AND WORD EMBEDDING MATRIX

As a first step, we import a vocabulary into the model. The vocabulary used for training large language models (LLMs) typically consists of a mix of general and domain-specific terms, including basic vocabulary, technical terminology, academic and formal language, idiomatic expressions, cultural references, as well as synonyms and antonyms. Each word and character are stored in a word lookup table and assigned a unique token. This process is called tokenization.

Many LLMs use Byte Pair Encoding (BPE), which splits words into subword units. For example, the word "*unhappiness*" might be broken down into "*un*," "*happi*," and "*ness*." BPE is widely used because it effectively balances vocabulary size and tokenization efficiency, particularly for handling rare words and sub-words. For simplicity, we use complete words in all our examples.

Figure 7-1 illustrates the relationship between words in the vocabulary and their corresponding tokens. Token values start from 2 because token 0 is reserved for padding and token 1 for unknown words.

Each token, representing a word, is mapped to a Word Embedding Vector, which is initially assigned random values. The collection of these vectors forms a Word Embedding Matrix.

The dimensionality of each vector determines how much contextual information it can encode.

For example, consider the word “clear.” A two-dimensional vector may distinguish it as either an adjective or a verb but lacks further contextual information. By increasing the number of dimensions, the model can capture more context and better understand the meaning of the word. In the sentence “The sky was clear,” the phrase “The sky was” suggests that "clear" is an adjective. However, if we extend the sentence to “She decided to clear the backyard of junk,” the word "clear" now functions as a verb. More dimensions allow the model to utilize surrounding words more effectively for next-word prediction. For instance, GPT-3 uses 12,288dimensional vectors. Given a vocabulary size of 50,000 words used by GPT-3, the Word Embedding Matrix has dimensions of 12,288 × 50,000, resulting in 614,400,000 parameters.

The context size, defined as the sequence length of vectors, determines how many preceding words the model considers when predicting the next word. In GPT-3, the context size is 2,048 tokens.

Figure 7-1: *Tokenization and Word Embedding Matrix.*

Word Embedding

As a first step, when we feed input words into a Natural Language Processing (NLP) model, we must convert them into a format the model can understand. This is a two-step process:

- **Tokenization:** Each word is assigned a corresponding token from a lookup table.
- **Word Embedding:** These token IDs are then mapped to vectors using a word embedding lookup table.

To keep things simple, Figure 7-2 uses two-dimensional vectors in the embedding matrix. Instead of complete sentences, we use words, which can be categorized into four groups: female, male, adult, and child.

The first word, "Wife," appears in the lookup table with the token value 2. The corresponding word vector in the lookup table for token 2 is [-3.5, -3.0]. The second word, "Mother," is assigned the token 3, which is associated with the word vector [-2.5, +2.0], and so on.

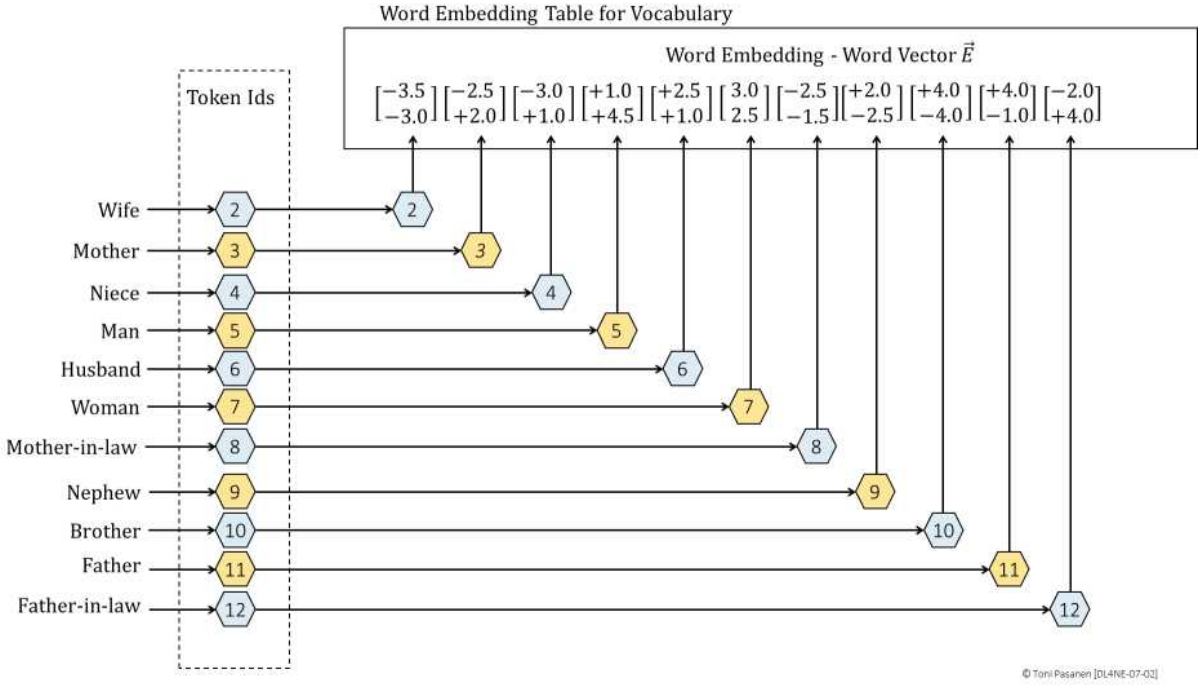


Figure 7-2: *Word Tokenization and Word Embedding*.

In Figure 7-3, we have a two-dimensional vector space divided into four quadrants, representing gender (male/female) and age (child/adult). Tokenized words are mapped into this space.

At the start of the first iteration, all words are placed randomly within the two-dimensional space. During training, our goal is to adjust the word vector values so that adults are positioned on the positive side of the Y-axis and children on the negative side. Similarly, males are placed in the negative space of the X-axis, while females are positioned on the positive side.

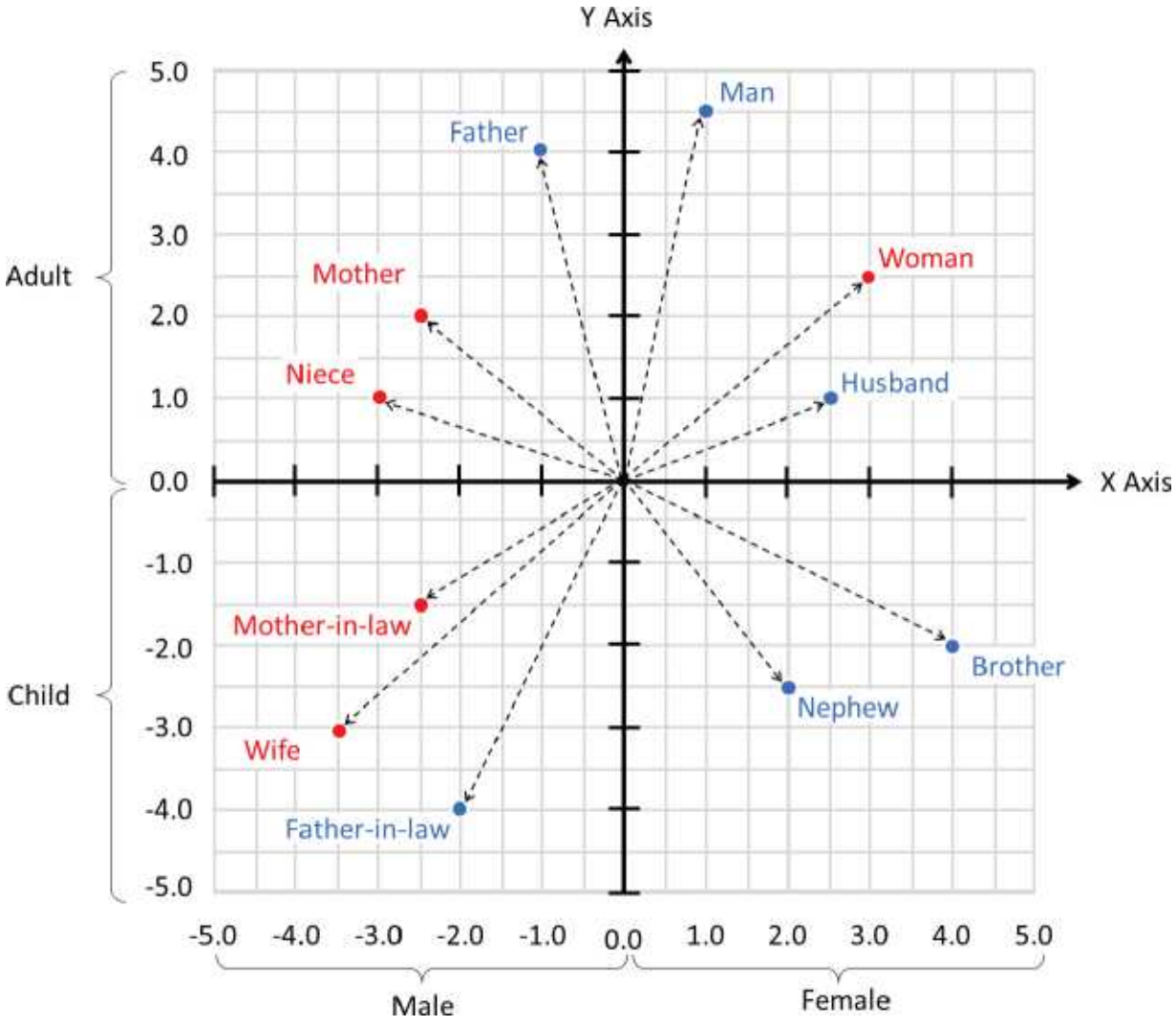


Figure 7-3: *Words in the 2 Dimensional Vector Space in the Initial State*.

Figure 7-4 illustrates how words may be positioned after successful training. All words representing a male adult are placed in the upperleft quadrant (adult/male). Similarly, all other words are positioned in the two-dimensional vector space based on their corresponding age and gender.

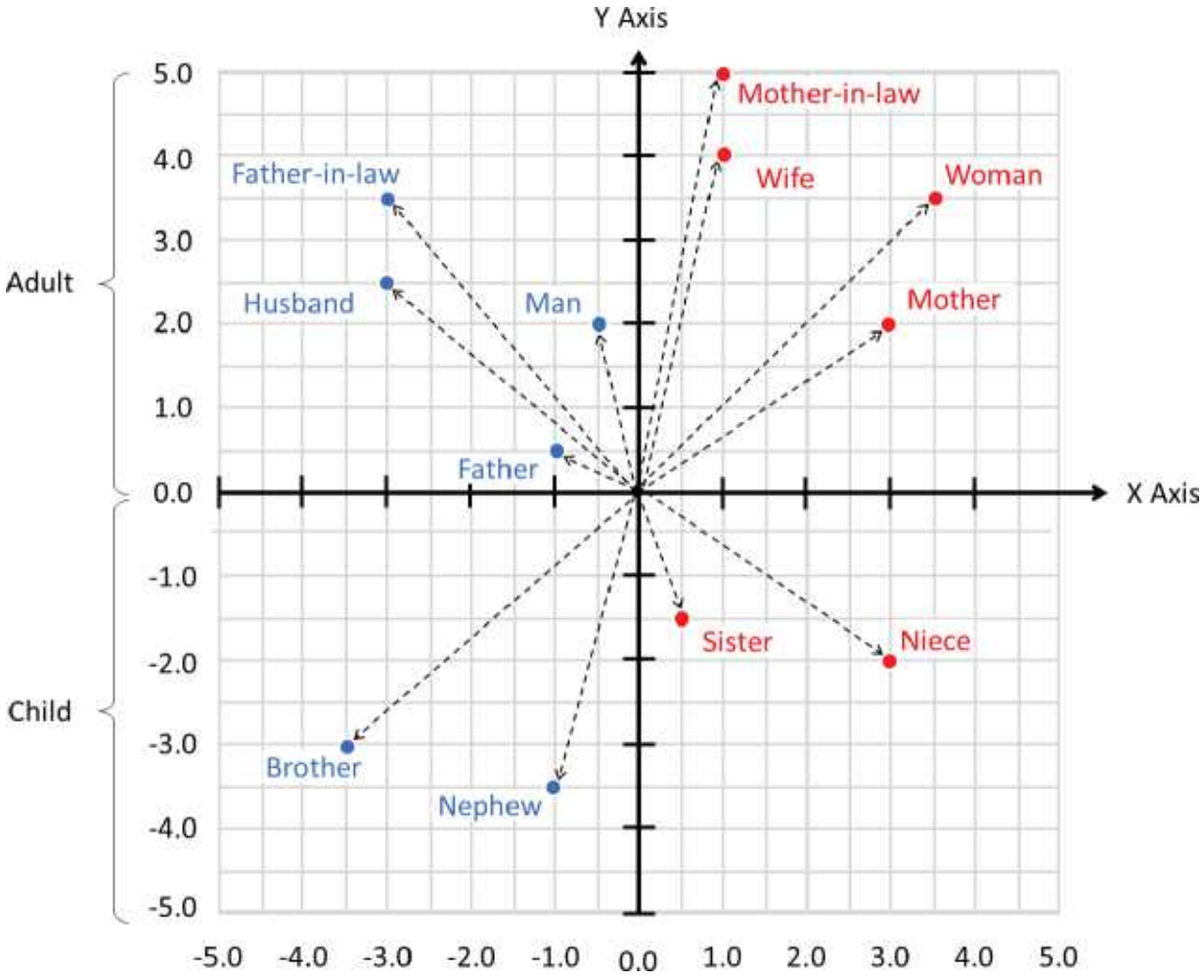


Figure 7-4: *Words in the 2 Dimensional Vector Space After Training.*

In addition to grouping similar words, such as "adult/female," close to each other in an n-dimensional space, there should also be positional similarities between words in different quadrants. For example, if we calculate the Euclidean distance between the words *Father* and *Mother*, we might find that their distance is approximately 4.3. The same pattern applies to word pairs like *Nephew*-*Niece*, *Brother*-*Sister*, *Husband*-*Wife*, and *Father-in-Law*–*Mother-in-Law*.

However, it is important to note that this example is purely theoretical. In practice, Euclidean distances in high-dimensional word embeddings are not fixed but vary depending on the training data and optimization process. The relationships between words are often better captured through cosine similarity rather than absolute Euclidean distances.

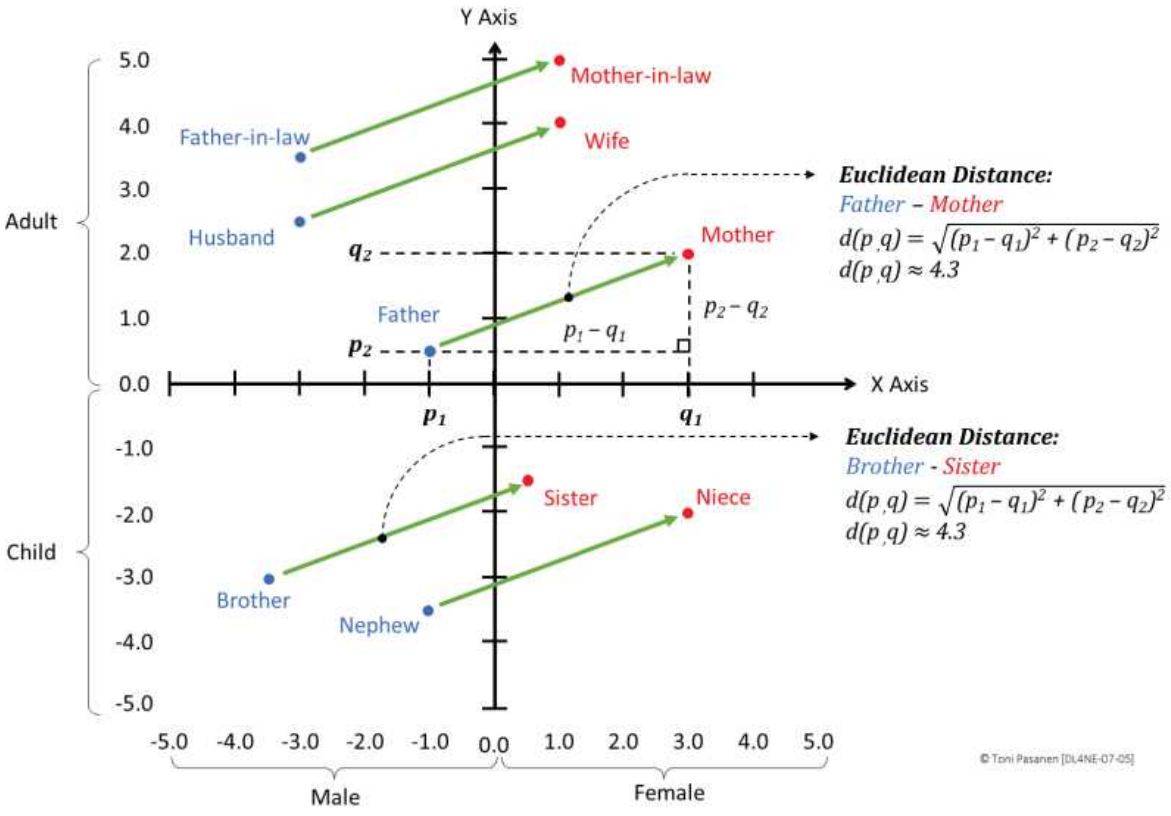


Figure 7-5: *Euclidean Distance.*

Positional Embeddings

Since input text often contains repeated words with different meanings depending on their position, an LLM must distinguish between them. To achieve this, the word embedding process in Natural Language Processing (NLP) incorporates a Positional

Encoding Vector alongside the Word Embedding Vector, resulting in the final word representation.

In Figure 7-6, the sentence "The sky is clear, so she finally decided to clear the backyard" contains the word *clear* twice. Repeated words share the same token ID instead of receiving unique ones. In this example, *the* is assigned token ID 2, and *clear* is assigned 5. These token IDs are then mapped to vectors using a word embedding lookup table. However, without positional encoding, words with different meanings would share the same vector representation.

Focusing on *clear* (token ID 5), it maps to the word embedding vector [+2.5, +1.0] from the lookup table. Since token IDs do not capture word position, identical words always receive the same embedding.

Positional encoding is essential for capturing context and semantic meaning. As shown in Figure 7-6, each input word receives a Positional Encoding Vector (PE) in addition to its word embedding. PE can either be learned and adjusted during training or remain fixed. The final Word Embedding Vector is computed by combining both the Word Embedding Vector and Positional Encoding Vector.

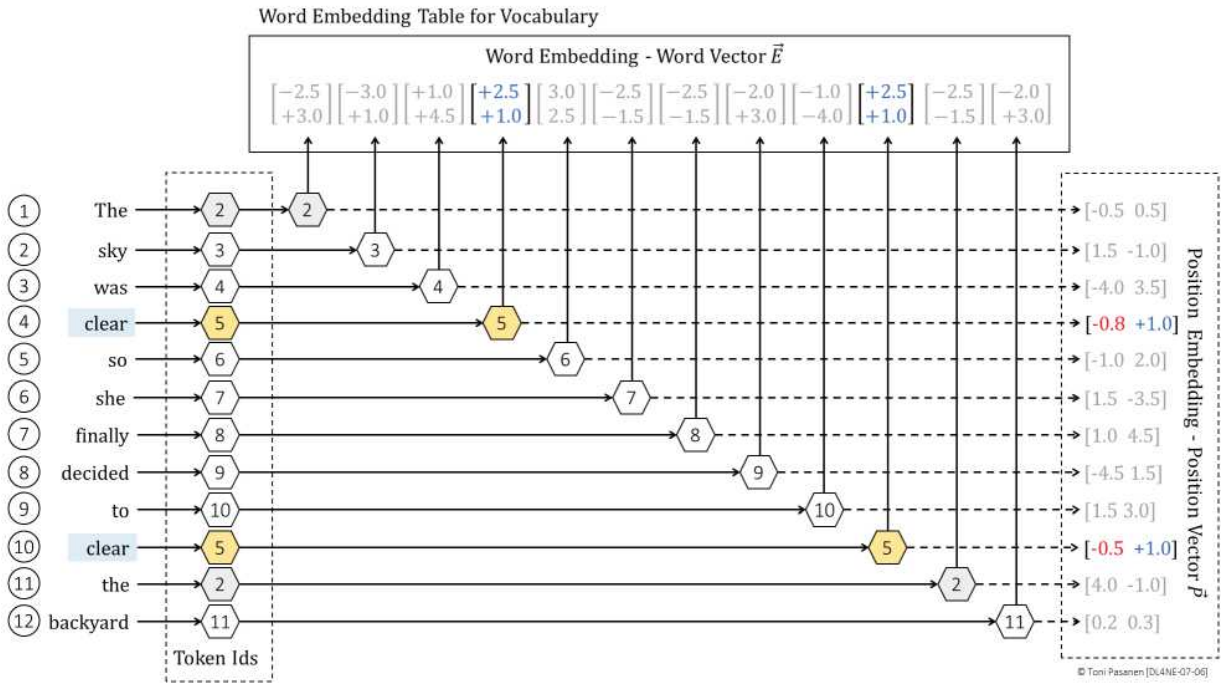


Figure 7-6: *Tokenization – Positional Embedding Vector.*

Calculating the Final Word Embedding

Figure 7-7 presents the equations for computing the final word embedding by incorporating positional embeddings. There are three variables:

- **Position (pos)** → The word's position in the sentence. In our example, the first occurrence of *clear* is the fourth word, so **pos = 4**.
- **Dimension (d)** → The depth of the vector. We use a **2-dimensional vector**, so **d = 2**.
- **Index (i)** → Specifies the axis of the vector: **0** for the x-axis and **1** for the y-axis.

The positional embedding is computed using the following equations:

- **x-axis:** $\sin(pos/10000^{2i/d})$, where i = 0
- **y-axis:** $\cos(pos/10000^{2i/d}$, where i = 1

For *clear* at position **4**, with **d = 2**, the resulting 2D positional vector is **[-0.8,+1.0]**. This vector is then added to the input word embedding vector **[+2.5, +1.0]**, resulting in the final word embedding vector **[+1.7, +2.0]**.

Figure 7-7 also shows the final word embedding for the second occurrence

of *clear*, but the computation is omitted.

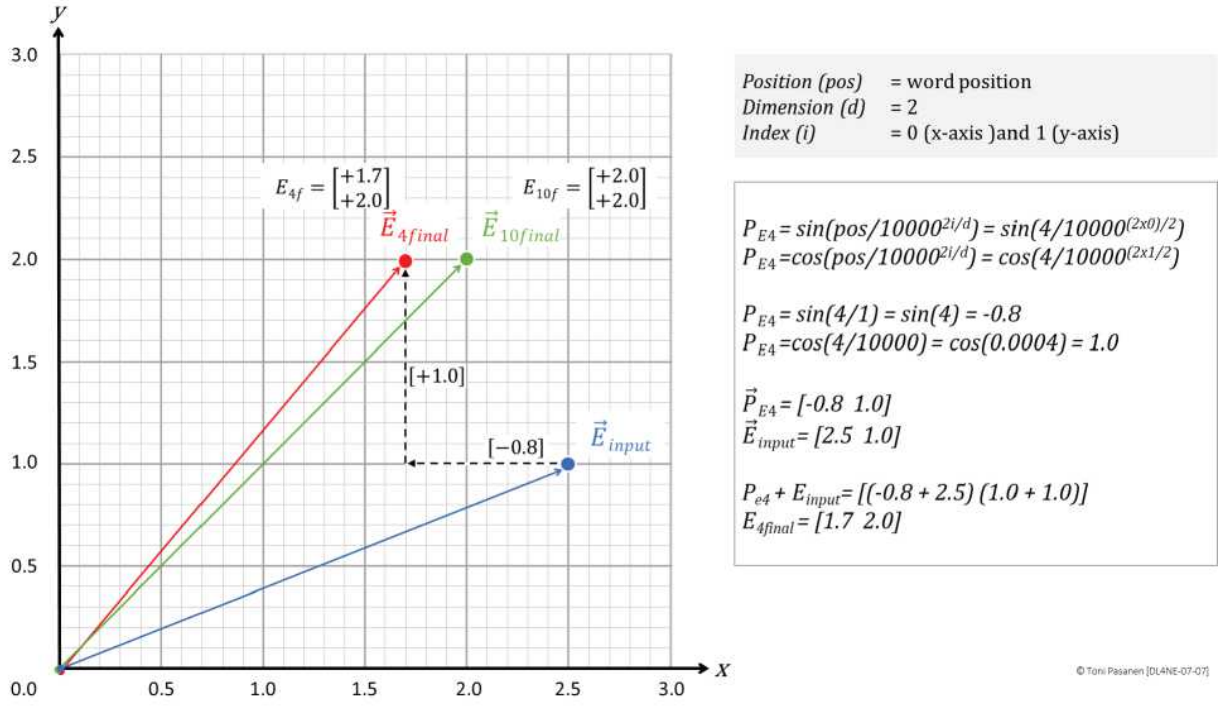


Figure 7-7: *Finals Word Embedding for the 4th Word.*

TRANSFORMER ARCHITECTURE

Introduction

Sequence-to-sequence (seq2seq) language translation and Generative Pretrained Transformer (GPT) models are subcategories of Natural Language Processing (NLP) that utilize the Transformer architecture. Seq2seq models are typically using Long Short-Term Memory (LSTM) networks or encoder-decored based Transformers. In contrast, GPT is an autoregressive language model that uses decoder-only Transformer mechanism. The purpose of this chapter is to provide an overview of the decoder-only Transformer architecture.

The Transformer consists of stacks of decoder modules. A word embedding vector, a result of the word tokenization and embbeding, is fed as input to the first decoder module. After processing, the resulting context vector is passed to the next decodeer, and so on. After the final decoder, a softmax layer evaluates the output against the complete vocabulary to predict the next word. As an autoregressive model, the predicted word vector from the softmax layer is converted into a token before being fed back into the subsequent decoder layer. This process involves a token-toword vector transformation prior to re-entering the decoder.

Each decoder module consists of an *attention layer*, *Add & Normalization layer* and a *feedforward neural network* (FFNN). Rather than feeding the embedded word vector (i.e., token embedding plus positional encoding) directly into the decoder layers, the Transformer first computes the *Query* (Q), *Key* (K), and *Value* (V) vectors from the word vector. These vectors are then used in the self-attention mechanism. Initially, the query vector is multiplied by the key vectors using matrix multiplication. The result is then divided by the square root of the dimension of the key vectors (scaled dot product) to obtain the logits. The logits are processed by a softmax layer to compute probabilities. The SoftMax prediction results are multiplied with the value vectors to produce a *context vector*.

Before feeding the context vector into the feedforward neural network, it is summed with the original word embedding vector (which includes positional encoding) via a residual connection. Finally, the output is normalized using layer normalization. This normalized output is then passed as input to the FFNN, which computes the output.

The basic architecture of the FFNN in the decoder is designed so that the input layer has as many neurons as the dimension of the context vector. The hidden layer, in turn, has four times as many neurons as the input layer, while the output layer has the same number of neurons as the input layer. This design guarantees that the output vector of the FFNN has the same dimension as the context vector. Like the attention block, the

FFNN block also employs residual connections and normalization.

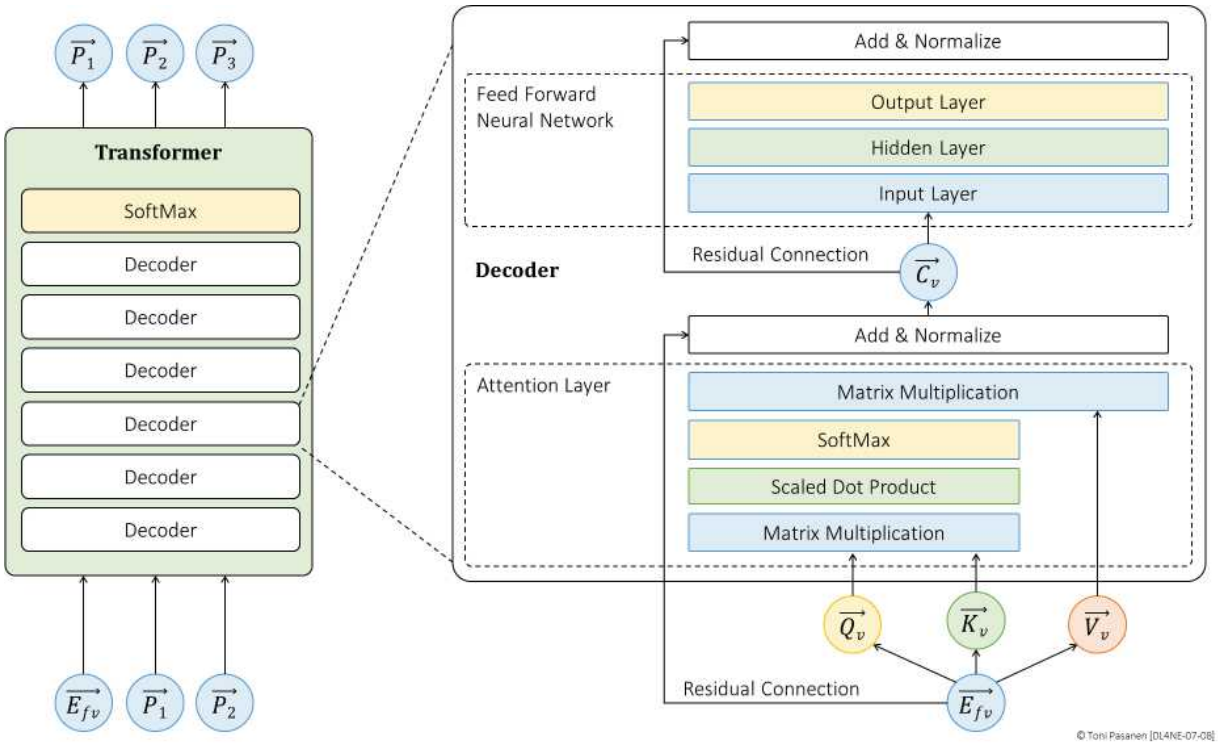


Figure 7-8: *Decoder-Only Transformer Architecture*.

Query, Key and Value Vectors

As pointed out in the Introduction, the word embedding vector is not used as input to the first decoder. Instead, it is multiplied by pretrained Query, Key, and Value weight matrices. The result of this matrix multiplication, dot product, produces the Query, Key, and Value vectors, which are use as inputs, and are processed through the Transformer. Figure 7–9 show the basic workflow of this process.

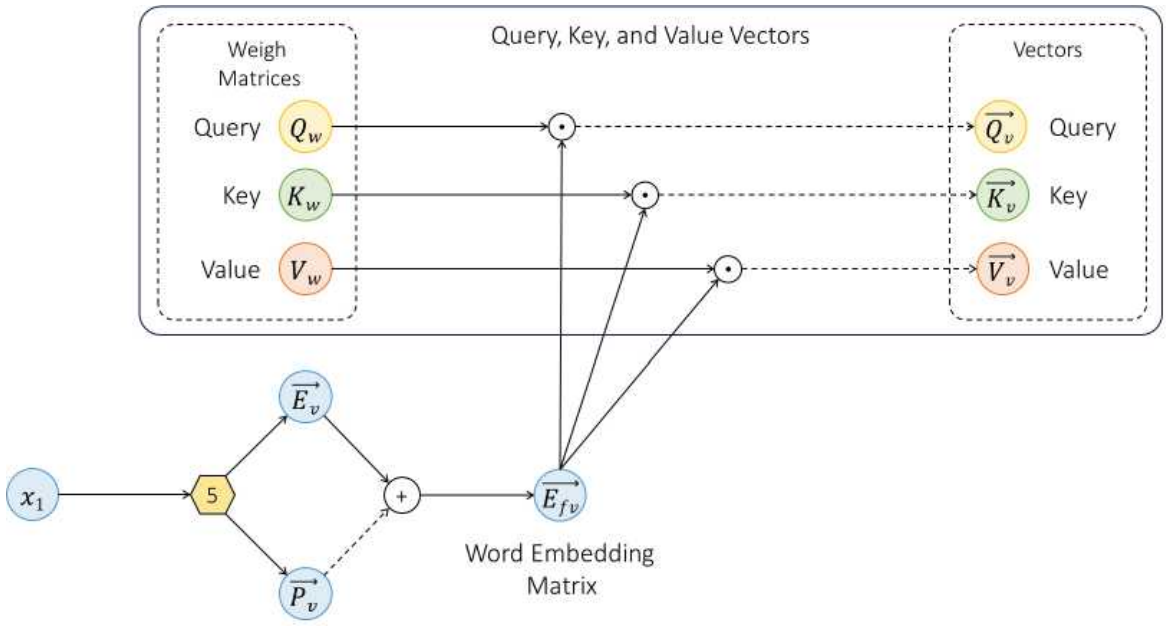


Figure 7-9: *Query, Key, and Value Vectors.*

Let's take a closer look at the process using numbers. After tokenizing the input words and applying positional encoding, we obtain a final 5dimensional word matrix. To reduce computation cycles, the process reduces the dimension of the Query vector from 5 to 3. Because we want the Query vector to be three-dimensional, we use three 5-dimensional column vectors, each of which is multiplied by the word vector.

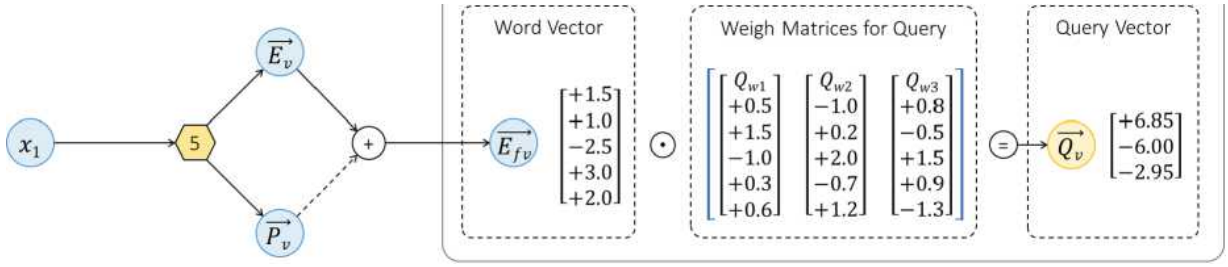


Figure 7-10: *Calculating the Query Vector.*

Figure 7–11 depicts the calculation process, where each component of the word vector is multiplied by its corresponding

component in the Query weight matrix. The weighted sum of these results forms a threedimensional Query vector. The Key and Value vectors are calculated using the same method. The same Query, Key, and Value (Q, K, V) weight matrices are used across all words (tokens) within a single self-attention layer in a Transformer model. This ensures that each token is processed in the same way, maintaining consistency in the attention computations. However, each decoder layer in the Transformer has its own dedicated Q, K, and V weight matrices, meaning that every layer learns different transformations of the input tokens, allowing deeper layers to capture more abstract representations.

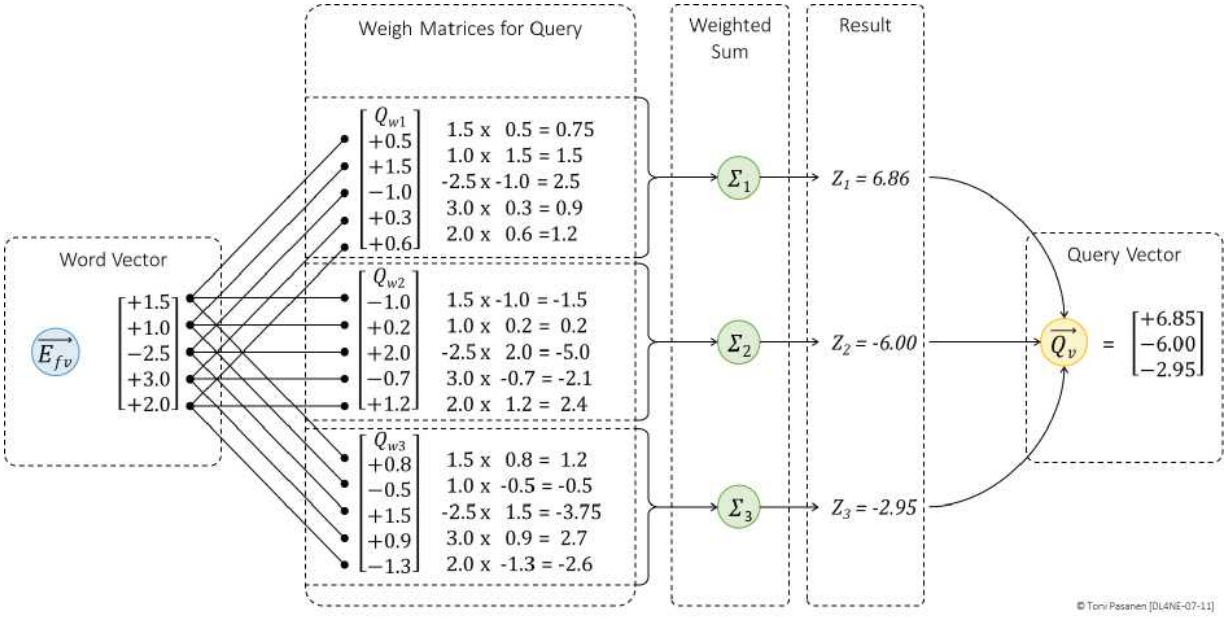


Figure 7-11: *Calculating the Query Vector.*

Attention Layer

Figure 7-12 depicts what happens in the first three components of the Attention layer after calculating the Query, Key, and Value

vectors. In this example, we focus on the word “clear”, and try to predict the next word. Its Query vector is multiplied by its own Key vector as well as by all the Key vectors generated for the preceding words. Each multiplication produces its own score. Note that the score values shown in the figure are theoretical and are not derived from the actual $Qv \times Kv$ matrix multiplication; however, the remaining values are based on these calculations. Additionally, in our example, we use one-dimensional values (instead of actual vectors) to keep the figures and calculations simple. In reality, these are n-dimensional vectors.

After the $Qv \times Kv$ matrix multiplication, the resulting scores are divided by the square root of the vector depth, yielding logits, i.e., the input values for the softmax function. The softmax function then computes the exponential of each logit (using Euler’s number, approximately 2.71828) and divides each result by the sum of all exponentials. For example, the value 3.16, corresponding to the first word, is divided by 482.22, resulting in a probability of 0.007. Note that the sum of the probabilities is 1.0. Softmax ensures that the attention scores sum to 1, making them interpretable as probabilities and helping the model decide which input tokens to focus on when generating an output. In our example, the token for the word “clear” has the highest probability at this stage. The word “decided” has the second highest probability score (0.210), which indicates that the semantics of “clear”, which has the highest probability score (0.665), can be interpreted as an verb answering the question: “What she decided to do?

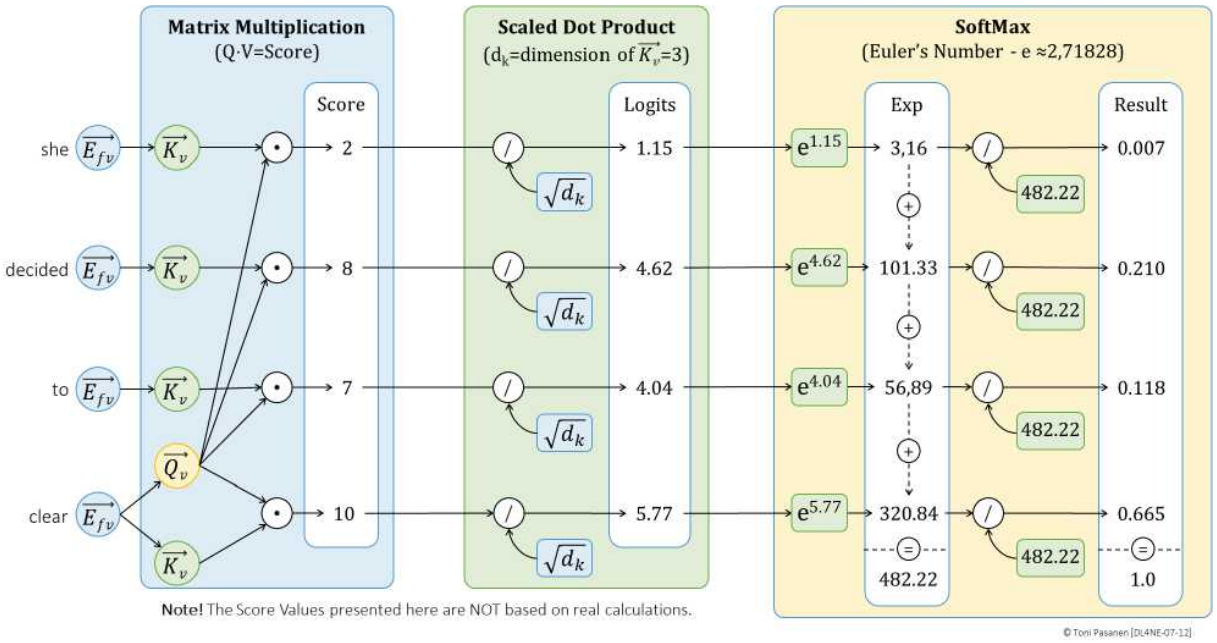


Figure 7-12: *Attention Layer, the First Three Steps.*

Next, the SoftMax probabilities are multiplied by each token's Value vector (matrix multiplication). The resulting vectors are then summed, producing the Context vector for the token associated with the word “clear.” Note that the components of the Value vectors are example values and are not derived from actual computations.

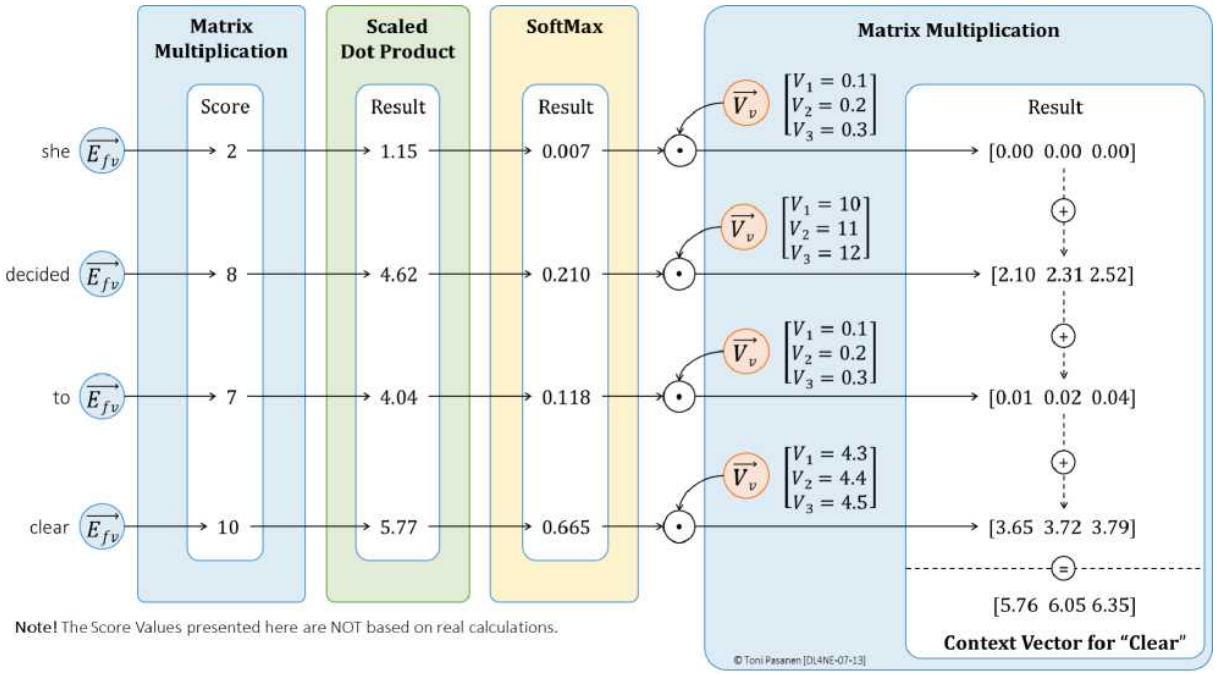


Figure 7-13: *Attention Layer, the Fourth Step.*

Add & Normalization

As the final step, the Word vector, which includes positional encoding, is added to the context vector via a Residual Connection. The result is then passed through a normalization process, where the vector's components are summed and divided by the vector's dimension, yielding the mean (μ). This mean value is then used for standard deviation calculation: the mean is subtracted from each of the three vector components, and the results are squared. These squared values are then summed, divided by three (the vector's dimension), and the square root of this result gives the final output vector [1.40, -0.55, -0.87] of the Add & Normalize layer.

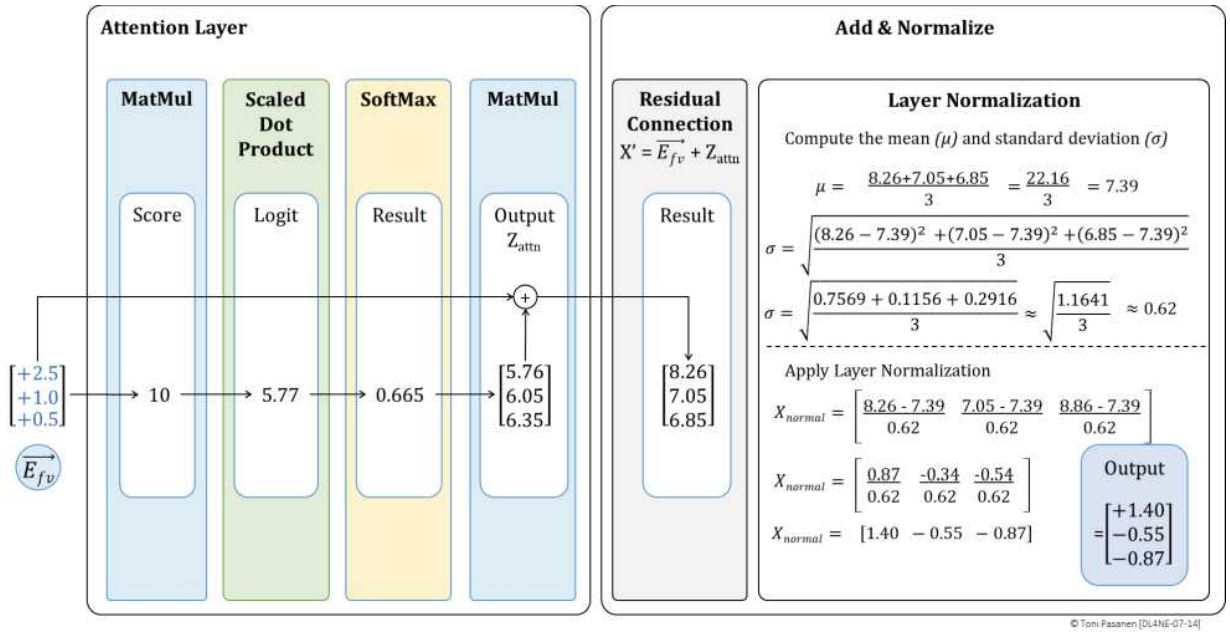


Figure 7-14: *Add & Normalize Layer – Residual Connection and Layer Normalization.*

Feed Forward Neural Network

Within the decoder module, the feedforward neural network uses the output vector from the Add & Normalize layer as its input. In our example, the FFNN have one neuron in input layer for each component of the vector. This layer simply passes the input values to the hidden layer, where each neuron first calculates a weighted sum and then applies the ReLU activation function. In our example, the hidden layer contains nine neurons (three times the number of input neurons). The output from the hidden layer is then fed to the output layer, where the neurons again compute a weighted sum and apply the ReLU activation function. Note that in transformer-based decoders, the FFNN is applied to each token individually. This means that

each token-related output from the attention layer is processed separately by the same FFNN model with shared weights, ensuring a consistent transformation of each token's representation regardless of its position in the sequence.

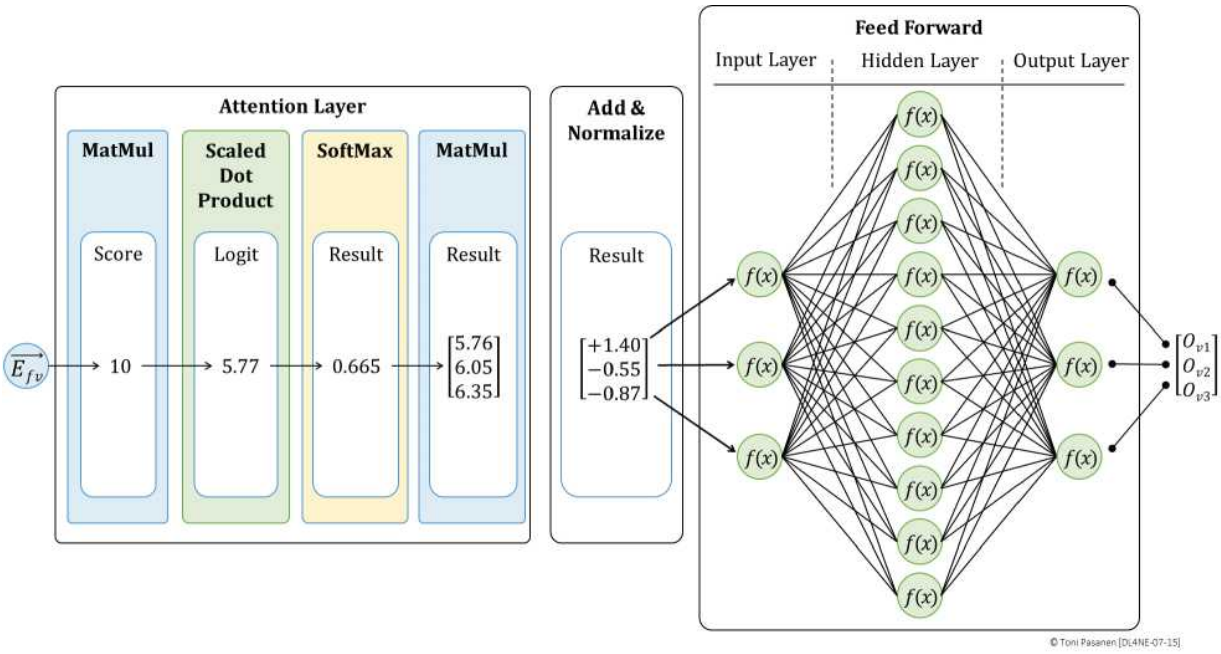


Figure 7-15: *Fully Connected Feed Forward Neural Network* (FFNN).

The final decoder output is computed in the Add & Normalize layer, similarly as Add & Normalize after the attention layer. This produces the decoder output, which is used as the input for the next decoder module.

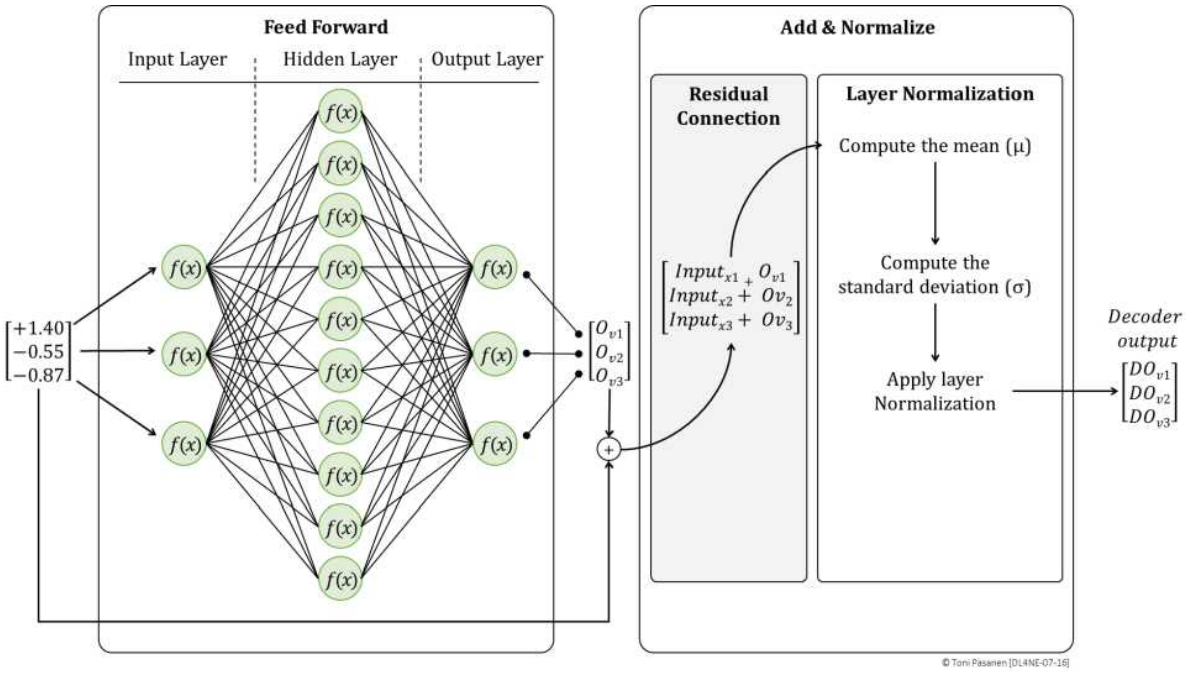


Figure 7-16: *Add & Normalize Layer – Residual Connection and Layer Normalization.*

Next Word Probability Computation – SoftMax Function

The output of the last decoder module does not directly represent the next word. Instead, it must be transformed into a probability distribution over the entire vocabulary. First, the decoder output is passed through a weight matrix that maps it to a new vector, where each element corresponds to a word in the vocabulary.

For example, in Figure 7-17 the vocabulary consists of 12 words. These words are tokenized and linked to their corresponding word embeddings vector. That said, the word embedding matrix serves as a weight matrix.

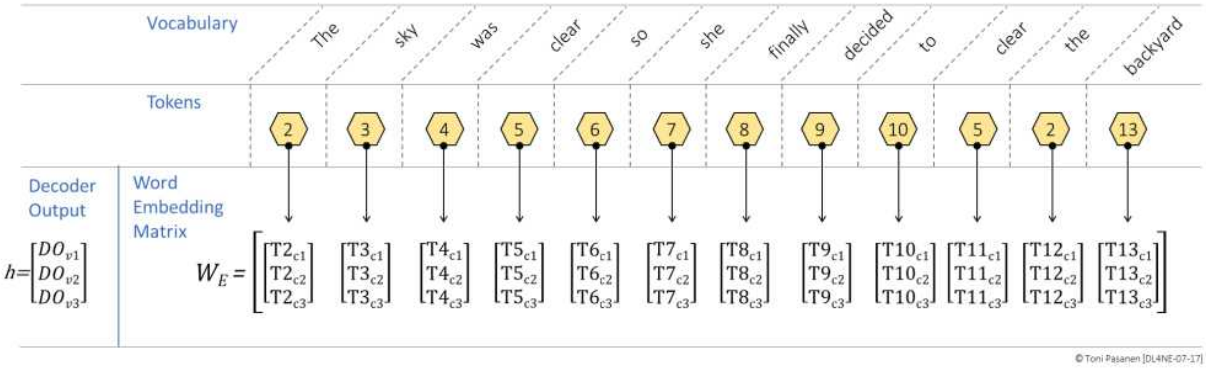


Figure 7-17: *Hidden State Vector and Word Embedding Matrix.*

Figure 7-18 illustrates how the decoder output vector (i.e., the hidden state *h*) is multiplied by all word embedding vectors to produce a new vector of logits.

Figure 7-18: *Logits Calculation – Dot Product of Hidden State and Complete Vocabulary.*

Next, the SoftMax function is applied to the logits. This function converts the logits into probabilities by exponentiating each logit and then normalizing by the sum of all exponentiated logits. The result is a probability distribution in which each value represents the likelihood of selecting a particular word as the next token.

Figure 7-19: *Probability Calculation - Adding Logits to SoftMax Function*

Finally, the word with the highest probability is selected as the next token. This token is then mapped back to its corresponding word using a token-to-word lookup. This initiates the next iteration, where the token is converted into its word embedding vector and used together with positional encoding to create the actual word embedding for the next iteration.

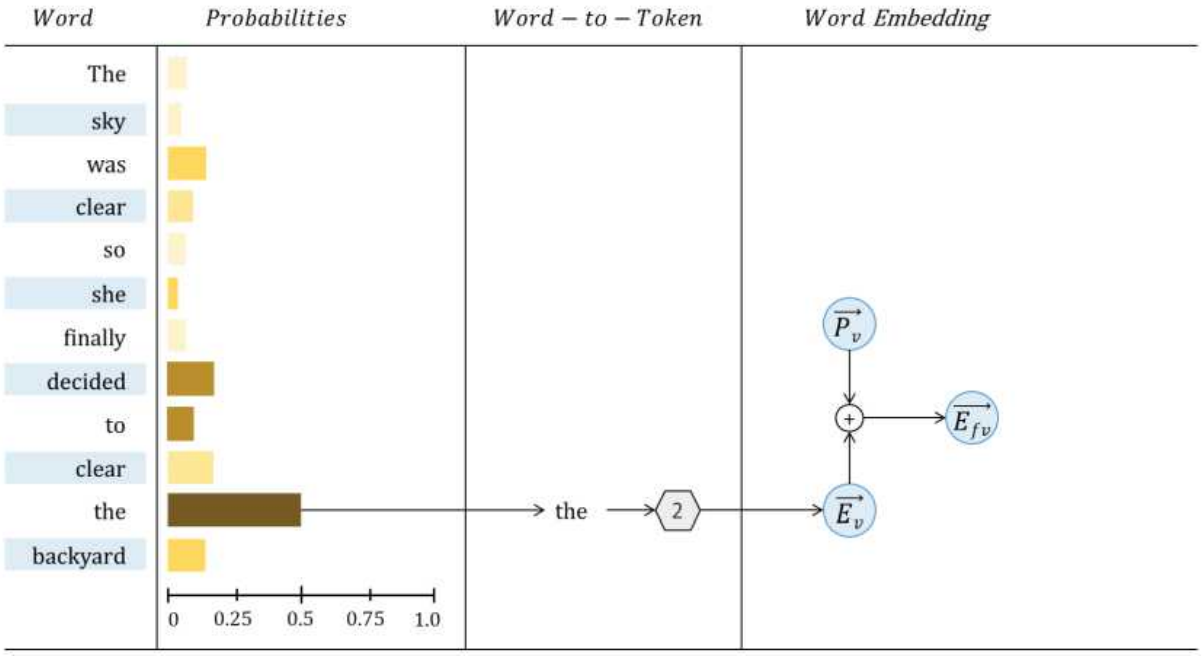


Figure 7-20: *Word-to-Token, and Token-to-Word Embedding Process.*

In theory, our simple example shows that the model can assign the highest probability to the correct word. For instance, by analyzing the position of the word “clear” relative to its preceding words, the model is able to infer the context. When the context implies that an action is directed toward a known target, the article “the” receives the highest probability score and is predicted as the next word.

CONCLUSION

We use pretty simple examples in this chapter. However, GPT-3, for example, is built on a deep Transformer architecture comprising 96 decoder blocks. Each decoder block is divided into three primary sublayers:

- Attention Layer: This layer implements multi-head self-attention using four key weight matrices, one each for the query, key, and value projections, plus one for the output projection. Together, these matrices account for roughly 600 million trainable parameters per block.
- Add & Normalize Layers: Each block employs two residual connections paired with layer normalization. The first Add & Normalize operation occurs immediately after the Attention Layer, and the second follows the Feed-Forward Neural Network (FFNN) layer. Although essential for stabilizing training, the parameters in each normalization step are relatively few, typically on the order of tens of thousands.
- Feed-Forward Neural Network (FFNN) Layer: The FFNN consists of two linear transformations with an intermediate expansion (usually about four times the model's hidden size). This layer contributes approximately 1.2 billion parameters per block.

Aggregating the parameters from all 96 decoder blocks, and including additional parameters from the token embeddings, positional encodings, and other components, the entire GPT-3 model totals around 175 billion trainable parameters. This is why parallelism is essential: the training process must be distributed across multiple GPUs and executed according to a selected parallelization strategy. The second part of the book discusses about Parallelization.

REFERENCES

[1] Magnus Ekman, “Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow”, 17 Aug. 2021

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017, June 12). Attention Is All You Need. arXiv.

https://arxiv.org/abs/1706.03762

[3] Google Developers. (2024, April). LLMs: What's a Large Language Model?

https://developers.google.com/machine-learning/crash-course/llm/transformers

[4] IBM. (2025, March 28). What is a Transformer Model?

https://www.ibm.com/think/topics/transformer-model

[5] TrueFoundry. (2023, March). Transformer Architecture in Large Language Models.

https://www.truefoundry.com/blog/transformer-architecture

CHAPTER 8: PARALLELISM STRATEGIES IN DEEP LEARNING

INTRODUCTION

Figure 8-1 depicts some of the model parameters that need to be stored in GPU memory: a) Weight matrices associated with connections to the preceding layer, b) Weighted sum (z), c) Activation values (y), d) Errors (E), e) Local gradients (local $\nabla\nabla$), f) Gradients received from peer GPUs (remote ∇), g) Learning rates (LR), and h) Weight adjustment values ($\Delta\Delta ww$). In addition, the training and test datasets, along with the model code, must also be stored in GPU memory. However, a single GPU may not have enough memory to accommodate all these elements. To address this limitation, an appropriate *parallelization strategy* must be chosen to efficiently distribute computations across multiple GPUs. This chapter introduces the most common strategies include Data Parallelism, Model Parallelism, Pipeline Parallelism, and Tensor Parallelism.

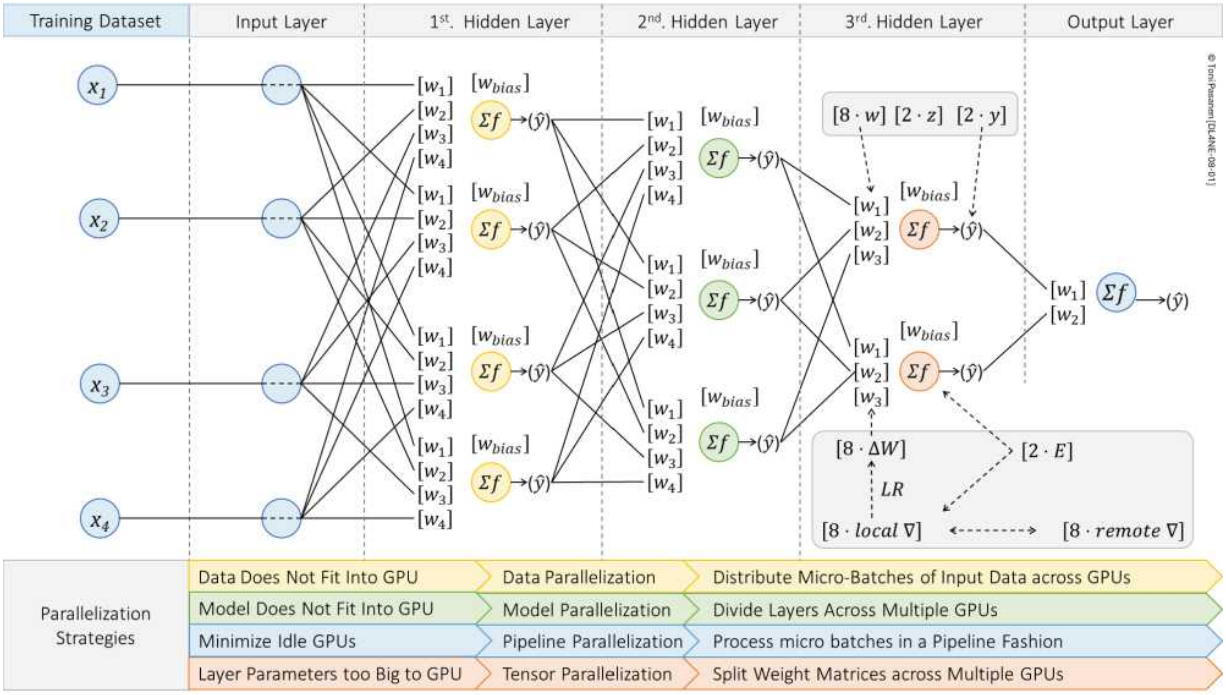


Figure 8-1: *Overview of Neural Networks Parameters.*

DATA PARALLELISM

In data parallelization, each GPU has an identical copy of the complete model but processes different mini-batches of data. Gradients from all GPUs are synchronized and averaged before updating the model. This approach is effective when the model fits within a single GPU's memory.

In Figure 8-2, the batch of training data is split into eight micro-batches. The first four micro-batches are processed by GPU A1, while the remaining four micro-batches are processed by GPU A2. Both GPUs share the same model, and their input data are processed through all layers to generate a model prediction. The computation during the forward pass does not involve network load traffic. After computing the model error, the backpropagation algorithm starts the backward pass. The first step involves calculating the derivative of the model error, which is synchronized across the GPUs. Next, the error is propagated backward to calculate neuron-based errors, which are then used to compute gradients for each weight parameter. These gradients are synchronized across the GPUs (Chapter 14 explains the process in detail). The backpropagation algorithm running on GPUs then sums the gradients and divides the result by the number of GPUs.

In our simple two-GPU example, this process does not generate excessive network traffic, although the GPUs can use

100% of their NICs forwarding capacity. However, if hundreds or even thousands of GPUs are used, the network traffic becomes significantly larger.

Inter-GPU network communication within a single server (using PCIe, NVLink) or between multiple servers (over InfiniBand, Ethernet, wireless) requires packet forwarding with minimal latency and in a lossless manner. Minimal latency is required to keep the training time as short as possible, while lossless transport is essential because training will pause if even a single packet is lost during synchronization. In the worst-case scenario, if no snapshot of the training progress is taken, the entire training process

must be restarted from the beginning. Training a Large Language Model can take months or more.

Now, consider the electricity costs if training had already been running for two months and had to be restarted due to a single packet loss.

Power Consumption Example:

- A single GPU consumes roughly 350W under full load.
- Total power consumption for 50,000 GPUs:

$350\ \text{W} \times 50{,}000 = 17{,}500{,}000\ \text{W} = 17.5\ \text{MW}$

- For two months (60 days = 1,440 hours) of training:

$17.5\ \text{MW} \times 1{,}440\ \text{hours} = 25{,}200\ \text{MWh} = 25{,}200{,}000$

• If electricity costs $0.10 per kWh, the total training cost will be: 25,200,000 kWh×0.10=2,520,000 USD

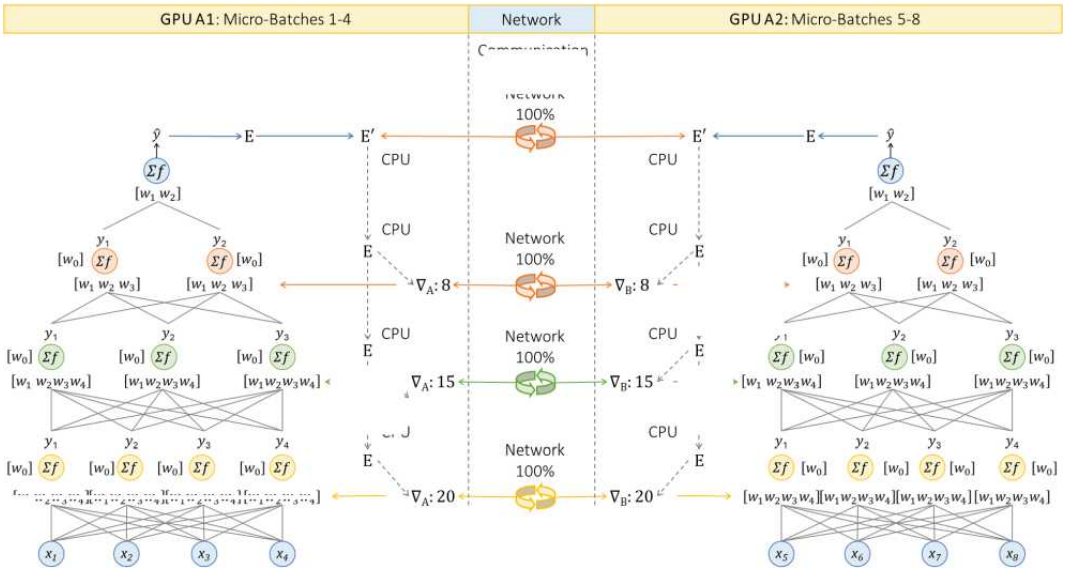


Figure 8-2: *Data Parallelism Overview.*

MODEL PARALLELISM WITH PIPELINE PARALLELISM

Model Parallelism is used when a neural network model is too large to fit into the memory of a single GPU. In this approach, different layers of the model are assigned to different GPUs. Each GPU is responsible for executing the computations, such as matrix multiplications and activation functions, associated with its designated layers. This allows the model to be trained across multiple GPUs without requiring the entire model to be loaded into each one.

Pipeline Parallelism, in turn, is a common implementation of model parallelism, further optimizes training by dividing each training batch into smaller *micro-batches*. These micro-batches are processed in a pipelined manner across the GPUs. While one GPU is working on a forward pass for one micro-batch, another GPU can start processing the next micro-batch, thereby increasing hardware utilization and throughput.

1st. Time Step – Active GPUs: 25% - Idle GPUs: 1a, 0b, 1b

In Figure 8-3, we have two GPU nodes, Host A and Host B, each equipped with two GPUs. The first hidden layer is initialized on GPU 0a, while the second hidden layer runs on GPU 1a. The third hidden layer and the output layer are placed on Host B, on GPUs

0b and 1b, respectively. This setup enables a layer-wise model parallelism strategy across four GPUs.

In this example, four mini-batches are processed sequentially by GPU 0a on Host A. At time step t_1, GPU 0a performs a matrix multiplication and applies an activation function to produce the first intermediate output, y_1. This result is stored in its local VRAM and then transferred to the VRAM of GPU 1a using Direct Memory Access (DMA) over a high-speed NVLink connection.

At this stage, the overall cluster GPU utilization is only 25%, since only GPU 0a is actively computing. Although data is being copied to GPU 1a, that GPU is not yet active because the DMA transfer bypasses the GPU's.

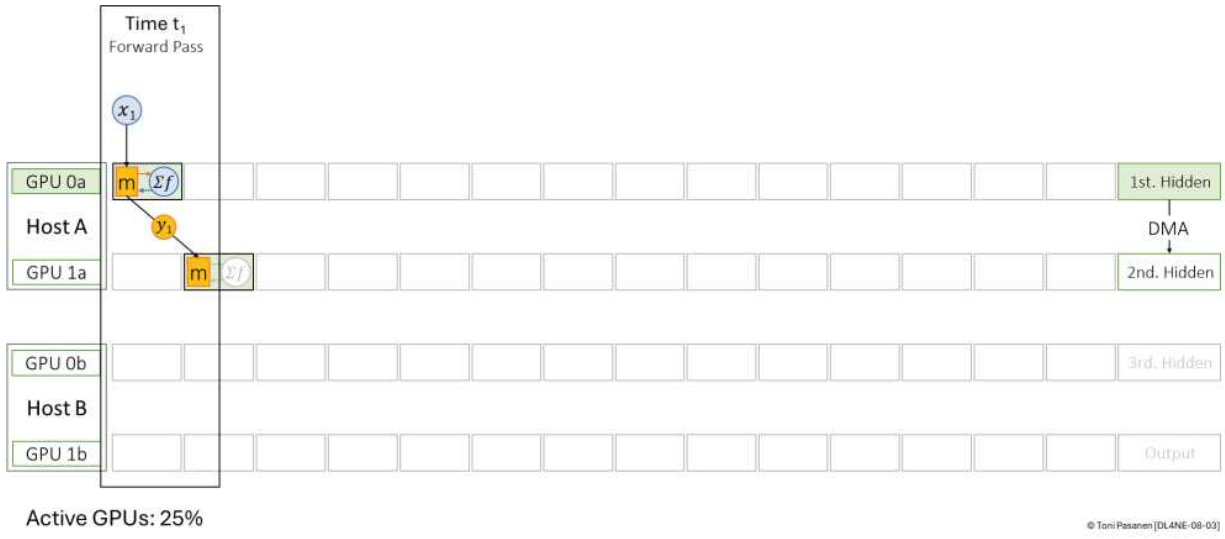


Figure 8-3: *Model Parallelism with Pipeline Parallelism – Time Step 1.*

2nd. Time Step — Active GPUs: 50% - Idle GPUs: 0b, 1b

At time step t_2, GPU 0a begins processing the second mini-batch by performing matrix multiplication and applying the activation function to produce output y_2. This output is stored in its local VRAM and then transferred to GPU 1a via DMA over NVLink, just like in the previous step.

Meanwhile, GPU 1a, which received y_1 during time step t_1, now becomes active. It reads y_1 from its VRAM, processes it through the second hidden layer, and produces an output. This result is then transferred to remote GPU 0b, which holds the third hidden layer. The transfer occurs over the backend Ethernet network using RoCEv2 (RDMA over Converged Ethernet version 2).

At this stage, GPUs 0a and 1a are actively computing, and GPU 0b is receiving data via RDMA. However, GPU 0b is not yet actively computing.

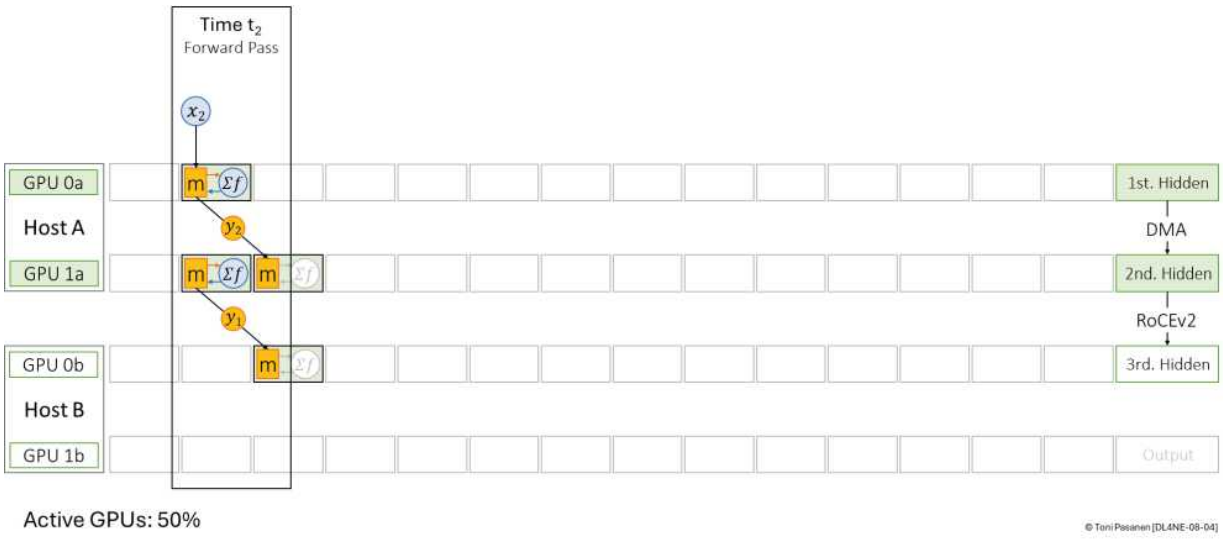


Figure 8-4: *Model Parallelism with Pipeline Parallelism – Time Step 2.*

3rd. Time Step— Active GPUs: 75% - Idle GPUs: 1b

At time step t_3, the pipeline continues to fill. GPU 0a begins processing the third mini-batch X3 by performing matrix multiplication and activation on the input, producing y_3. This output is stored in its local VRAM and transferred to GPU 1a over NVLink using DMA.

Simultaneously, GPU 1a processes the output y_2, received at t_2 from GPU 0a, through the second hidden layer. The resulting activation value is transferred over the backend network to GPU 0b on Host B.

At the same time, GPU 0b begins processing y_1 from its local VRAM. This intermediate result, originally received at t_2, is now passed through the third hidden layer. Once processed, the output is stored local VRAM, from where it is copied to VRAM of remote GPU 1b for final output layer computation.

All of these operations occur concurrently across the GPUs, effectively utilizing the pipeline. At this point, three out of the four GPUs are performing computation, raising the cluster's GPU utilization to 75%.

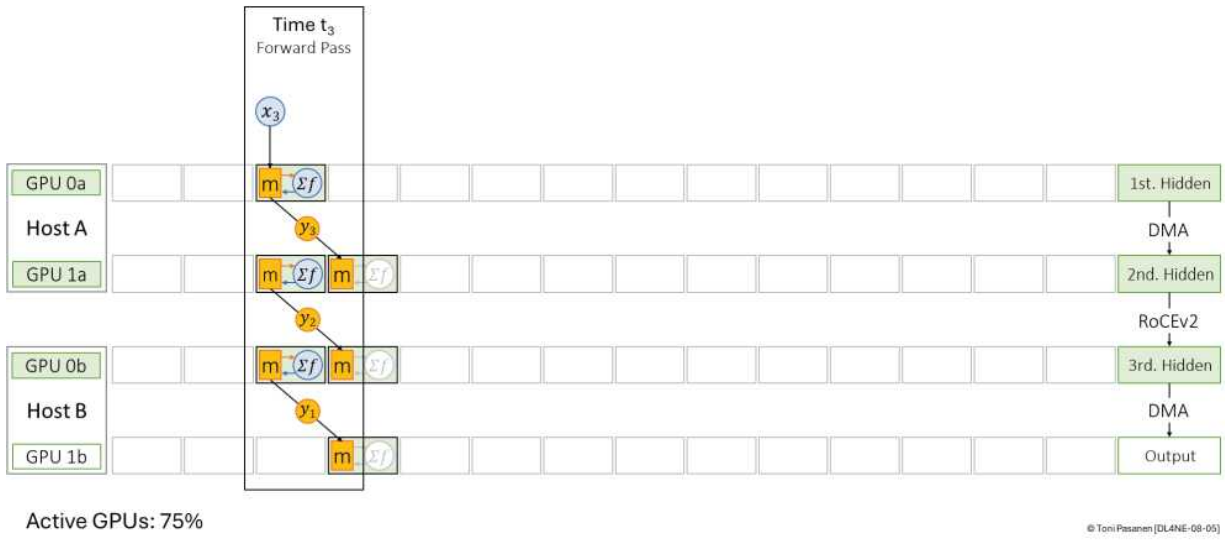


Figure 8-5: *Model Parallelism with Pipeline Parallelism – Time Step 3.*

4th. Time Step– Active GPUs: 100% - Idle GPUs: none

At time step t_4, the pipeline is fully active. GPU 0a processes the fourth mini-batch X4, producing output y_4 and transferring it to GPU 1a over NVLink. Simultaneously, GPU 1a processes y_3 and sends the result to GPU 0b on Host B over the backend Ethernet network using RoCEv2.

At the same time, GPU 0b processes y_2. Once completed, the output is transferred locally over NVLink to GPU 1b, which begins computing the final output layer for y_1, related to the first mini-batch X1.

At currently step t_4, all four GPUs are actively computing, and the pipeline reaches maximum throughput. GPU utilization is now 100%.

In addition to the forward computation, GPU 1b now initiates the backward pass related to the first mini-batch. It calculates the model error E_1 from the model output $yy^\wedge_1$ and propagates it backward to GPU 0b. At the same time, GPU 1b computes the weight adjustment values for the weight vectors related to first mini-batch X_1 and updates its local weights. These updated weights will then be used in the next iteration of the forward pass during matrix multiplication.

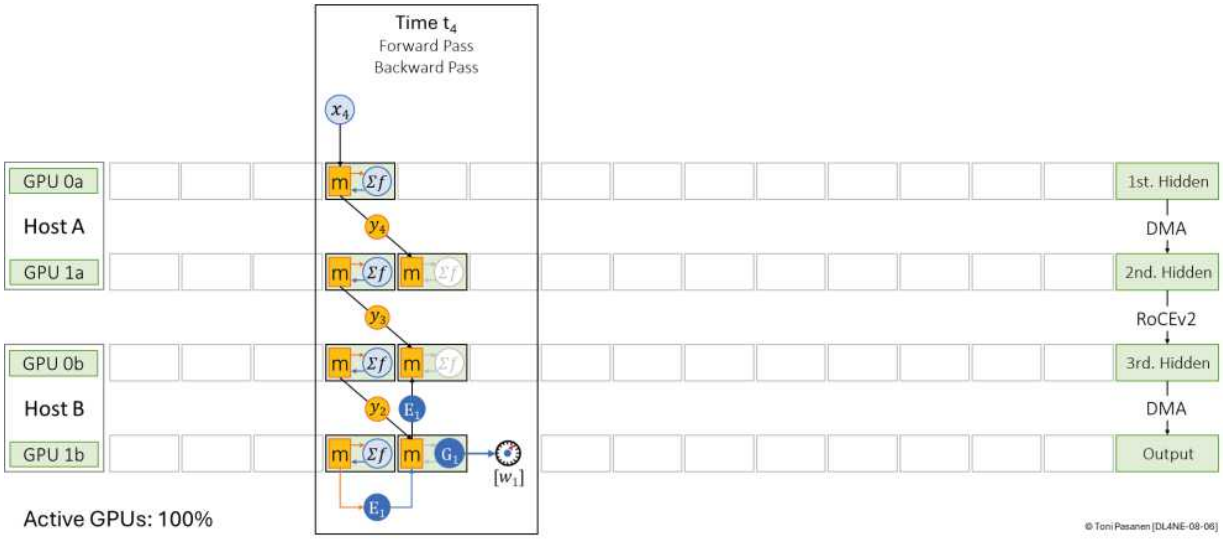


Figure 8-6: *Model Parallelism with Pipeline Parallelism – Time Step 4.*

5th. Time Step – Active GPUs: 75% - Idle GPUs: 0a

At time step t_5, the system continues forward computation while also propagating the backward pass.

Forward Pass: GPU 0a is now idle, having completed all four mini-batches. GPU 1a processes the output y_4, received from GPU 0a and transferring its output y_4 to GPU 0b using RoCEv2.

GPU 0b processes y_3 and sends the result over NVLink to GPU 1b. GPU 1b processes y_2 through the output layer.

Backward pass: After receiving the error E_1 in the previous time step, GPU 0b computes the gradient G_1, which is then used to calculate the weight update for the weight vectors related to the first mini-batch X_1. GPU 0b also propagates the error E_1 backward to GPU 1a. Simultaneously, GPU 1b performs the same process, computing the weight updates for the second mini-batch X_2. These updated weights will then be used in the next iteration of the forward pass during matrix multiplication.

At this point, the forward and backward passes start to overlap, with GPUs working on different mini-batches in both directions.

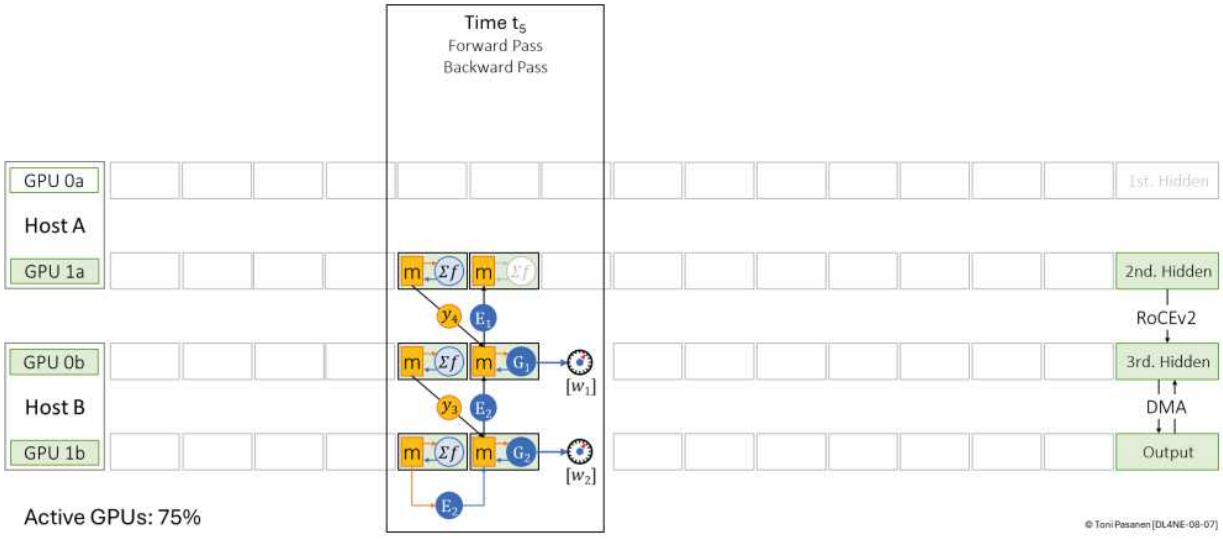


Figure 8-7: *Model Parallelism with Pipeline Parallelism – Time Step 5.*

Figures 8-8 through 8-12 illustrate how the backward pass progresses across all GPUs, showing how mini-batch-specific

errors are computed at each layer and used to update the weight matrices. These errors are then propagated backward all the way to GPU 0a.

Additionally, Figures 8-8 through 8-12 depict how GPU utilization changes across the time steps during the backward pass. As each GPU completes its respective tasks

6th. Time Step — Three Active GPUs, One Idle GPU: Overall GPU Utilization 75%

- GPU 1b computes new weights for the third mini-batch X_3.
- GPU 1a computes new weights for the second mini-batch X_2.
- GPU 1a computes new weights for the first mini-batch X_1.

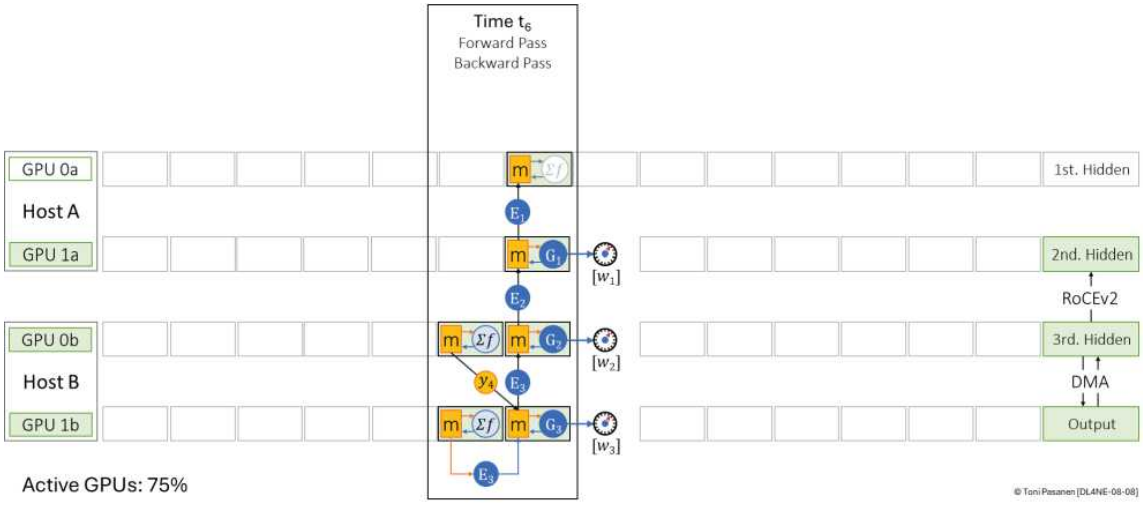


Figure 8-8: *Model Parallelism with Pipeline Parallelism – Time Step 6.*

7th. Time Step — Four Active GPUs: Overall GPU Utilization 100%

- GPU 1b computes new weights for the fourth mini-batch X_4.
- GPU 0b computes new weights for the third mini-batch X_3.
- GPU 1a computes new weights for the second mini-batch X_2.
- GPU 0a computes new weights for the first mini-batch X_1.

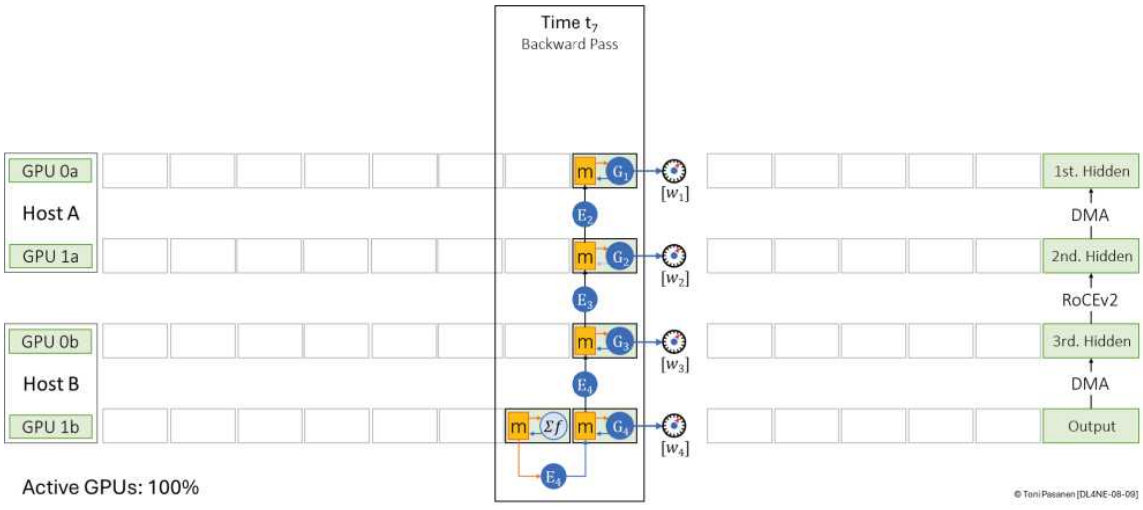


Figure 8-9: *Model Parallelism with Pipeline Parallelism – Time Step 7.*

8th. Time Step – Three Active GPUs: One Idle GPU: Overall GPU Utilization 75%

- GPU 0b computes new weights for the fourth mini-batch X_4.
- GPU 1a computes new weights for the third mini-batch X_3.
- GPU 0a computes new weights for the second mini-batch X_2.

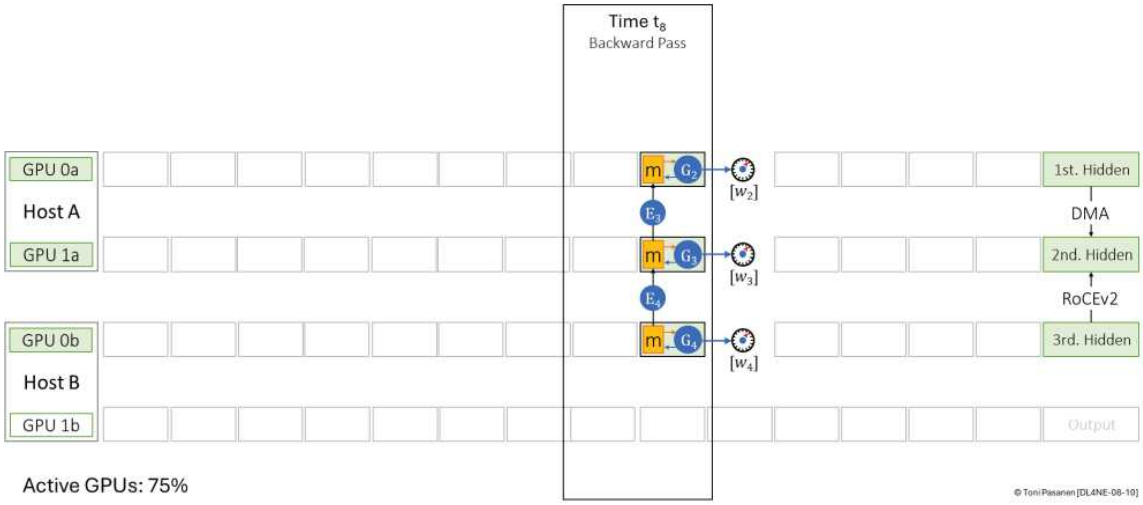


Figure 8-10: *Model Parallelism with Pipeline Parallelism – Time Step 8.*

9th. Time Step – Two Active GPUs: Two Idle GPUs: Overall GPU Utilization 50%

- GPU 1a computes new weights for the fourth mini-batch X_4.

- GPU 0a computes new weights for the third mini-batch X_3.

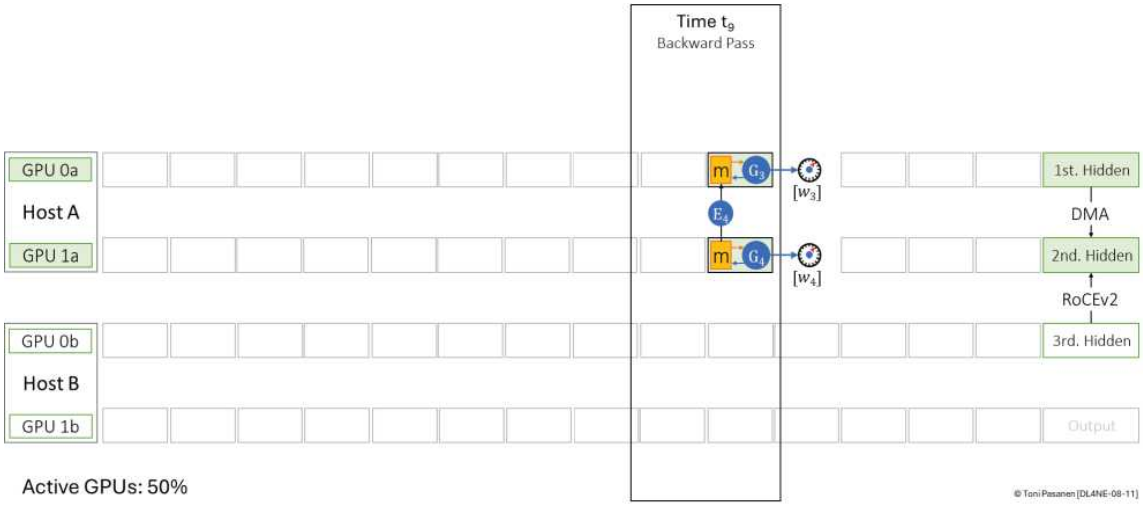


Figure 8-11: *Model Parallelism with Pipeline Parallelism – Time Step 9.*

10th. Time Step – One Active GPU: three Idle GPUs: Overall GPU Utilization 25%

- GPU 0a computes new weights for the fourth mini-batch X_4.

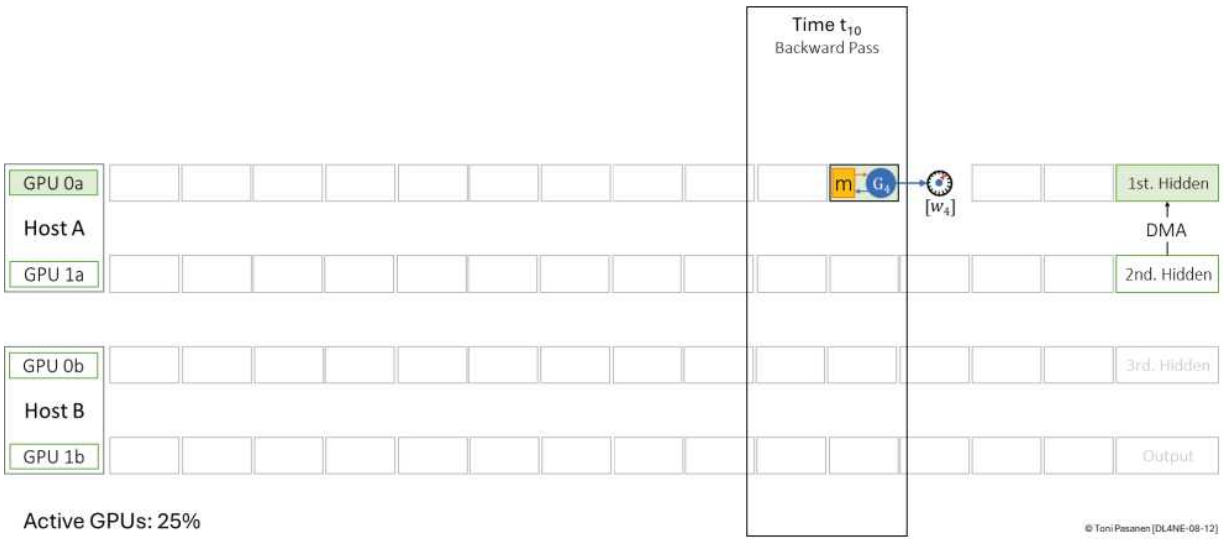


Figure 8-12: *Model Parallelism with Pipeline Parallelism – Time Step 10.*

After completing the backward pass process, the second iteration of forward pass can be initialized.

Figure 8-13 gives an overall view of pipelined model parallelization. This kind of illustration is often used to explain the concept, as it clearly shows the challenge of underutilized GPU resources. The white boxes indicate periods when the GPUs are idle. As you can see, the overall GPU utilization in the figure isn't great. There are several research efforts and practical solutions that can improve this and make training much more efficient. But the goal here is simply to explain the basic idea behind pipelined model parallelization. It's up to the reader to dig deeper into the topic if they're interested.

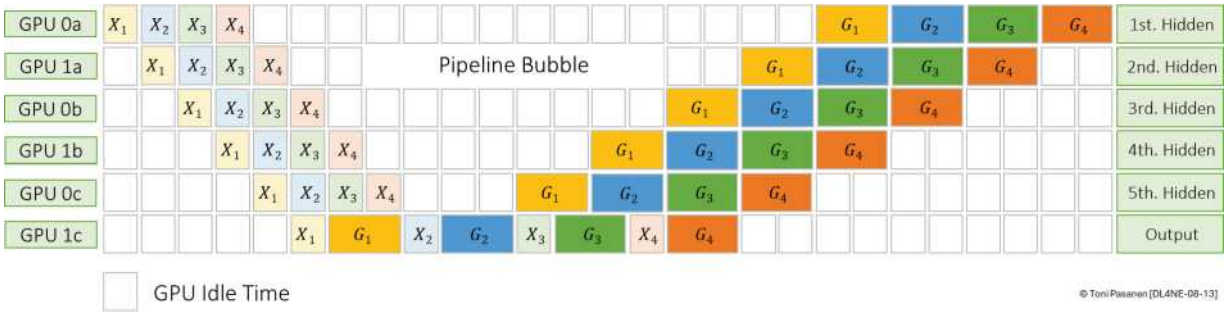


Figure 8-13: *Pipeline Bubble.*

TENSOR PARALLELISM

The previous section described how Pipeline Parallelism distributes entire layers across multiple GPUs. However, Large Language Models (LLMs) based on transformer architectures contain billions of parameters, making this approach insufficient.

For example, GPT-3 has approximately 605 million parameters in a single self-attention layer and about 1.2 billion parameters in a feedforward layer, and these figures apply to just one transformer block. Since GPT-3 has 96 transformer blocks, the total parameter count reaches approximately 173 billion. When adding embedding and normalization parameters, the total increases to roughly 175 billion parameters. The number of parameters in a single layer alone often exceeds the memory capacity of a single GPU, making Pipeline Parallelism insufficient. Additionally, performing large matrix multiplications on a single GPU would be extremely slow and

inefficient. Tensor Parallelism addresses this challenge by splitting computations within individual layers across multiple GPUs rather than assigning whole layers to separate GPUs, as done in Pipeline Parallelism.

Chapter 7 introduces Transformer architecture but for memory refreshing, figure 8-14 illustrates a stack of decoder

modules in a transformer architecture. Each decoder module consists of a Self-Attention layer and a Feedforward layer. The figure also shows how an input word, represented by *x1*, is first mapped to a token. The token, in turn, receives a positional word embedding vector through lookups in the word embedding and position embedding tables.

The resulting word vector is used to compute Query (Q) and Key (K) matrices, which, in turn, produces logits via dot products. These logits are then passed through the SoftMax function. The resulting matrix from the SoftMax function is multiplied with the Value (V) matrices. After Add & Normalization computation, the resulting matrix is fed into the Feedforward, fully connected, neural network.

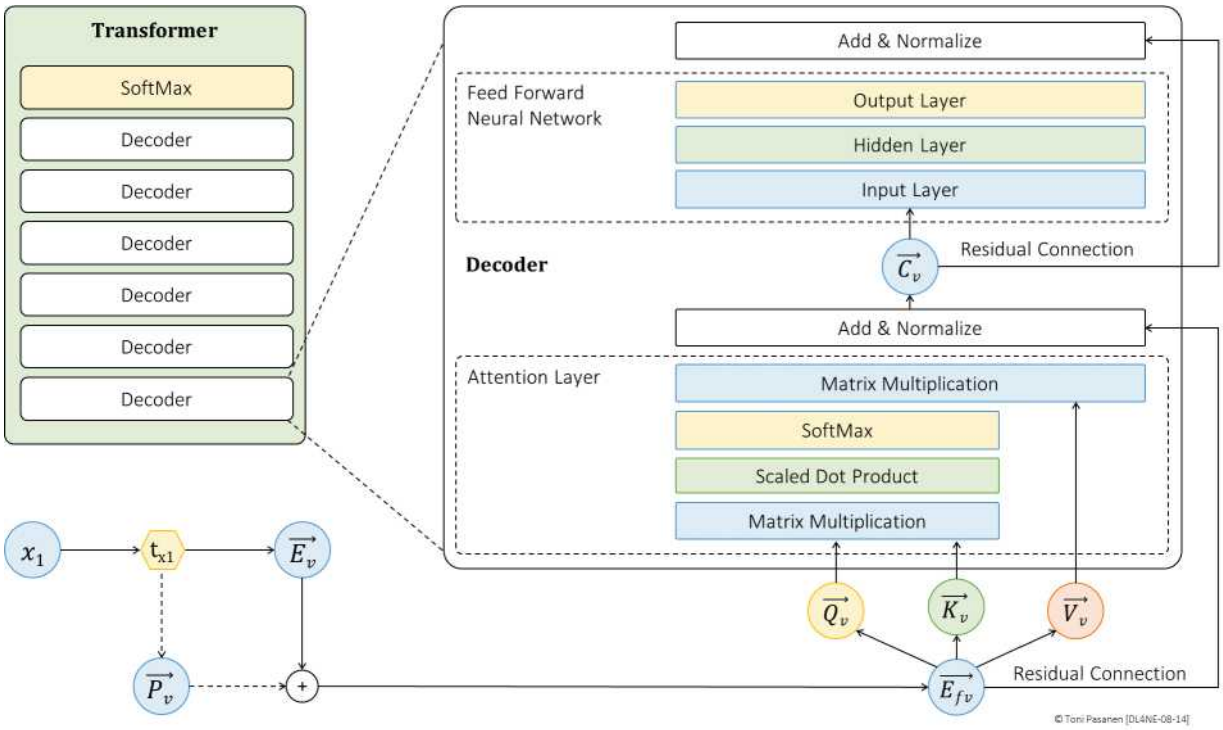


Figure 8-14: *An Overview of a Transformer Architecture.*

Self-Attention Layer

In most cases, the word embedding matrix fits within a single GPU. This is because a typical embedding matrix is approximately 200 MB, which is significantly smaller than large Transformer layers that can contain billions of parameters.

Another reason for keeping the embedding matrix on a single GPU is efficient lookup operations. Unlike large matrix multiplications, embedding lookups are memory-efficient and do not impose significant computational overhead. Splitting the embedding matrix across multiple GPUs would introduce high communication costs, as each GPU would store only a fraction of the vocabulary. This would require frequent crossGPU communication for token lookups, increasing latency and reducing efficiency. After the embedding lookup, the embedding vectors are broadcasted to all GPUs before the Transformer computations start.

However, in very large-scale models (such as GPT-3 with 175 billion parameters), embeddings may be sharded across multiple GPUs using distributed embeddings or model parallelism techniques. One approach is row-wise parallelism, where the vocabulary is split across GPUs, and each GPU stores only a fraction of the embeddings, handling lookups for the tokens it owns.

Figure 8-15 illustrates how the positional word embedding matrix (*Eepv*) is multiplied with the Query (Q), Key (K), and Value (V) matrices to produce the corresponding Q, K, and V

vectors. The Query and Key vectors are then used as inputs to the self-attention layer.

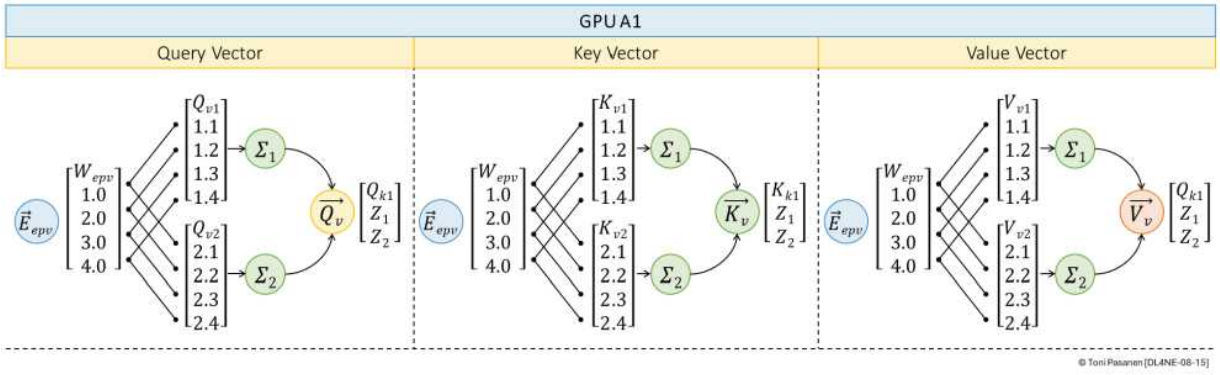


Figure 8-15: *Local Query* (Q), *Key* (K), *and Value* (V) *Matrices*.

Figure 8-16 illustrates how the Query, Key, and Value matrices are sharded across two GPUs. The first fragments of these matrices are assigned to GPU A1, while the second fragments are assigned to GPU A2. The positional word embedding matrix (*Eevp*) is also distributed between GPU A1 and GPU A2. Matrix multiplication is then performed between the corresponding fragment of *Eevp* and the respective shards of the Q, K, and V matrices.

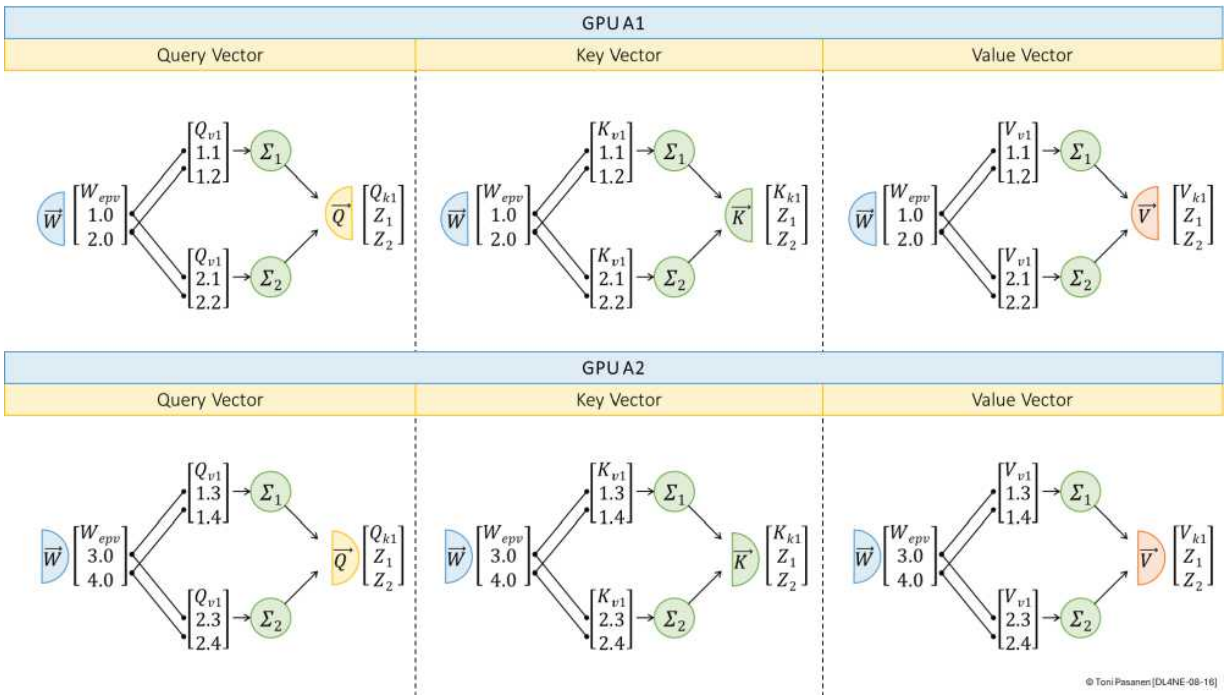


Figure 8-16: *Shared Query* (Q), *Key* (K), *and Value* (V) *Matrices*.

Figure 8-17 illustrates the cross-GPU communication involved in the forward pass of the Self-Attention layer when using Tensor Parallelism. In this example, both the word embedding, and positional embedding matrices fit within GPU A1. After computing the positional word embeddings for the input words, the resulting vectors are broadcasted to GPU A2.

Since we are using Tensor Parallelism, the Query (Q), Key (K), and Value (V) matrices are partitioned across GPU A1 and GPU A2. Once each GPU has computed its assigned slices of the Q, K, and V vectors, the Q and K vectors are shared between GPUs using an All-Gather operation. This ensures that each GPU receives the missing parts of the Q and K matrices, reconstructing the complete matrices across GPUs. Only the Q and K matrices are synchronized; the V matrix remains local to each GPU.

The Q and K matrices are then used in the Self-Attention layer, where the first operation is a matrix multiplication between the Query vectors and Key vectors for all tokens. The process is explained in detail in Chapter 7. The resulting scores are used to compute logits, which are inputs to the SoftMax function, using scaled dot-product attention. The output of the SoftMax function is then multiplied by the local fragment of the V matrix on each GPU.

The SoftMax operation produces a Context Vector (Cv) for each input word, which serves as the input to the Feedforward Neural Network (FFN) layer. That said, the SoftMax in the self-attention layer is not the final prediction layer, it's used to compute attention weights. The feedforward network processes the context vectors token representations produced by self-attention, not the predicted token. The final prediction is typically made by a separate output projection followed by a SoftMax over the vocabulary.

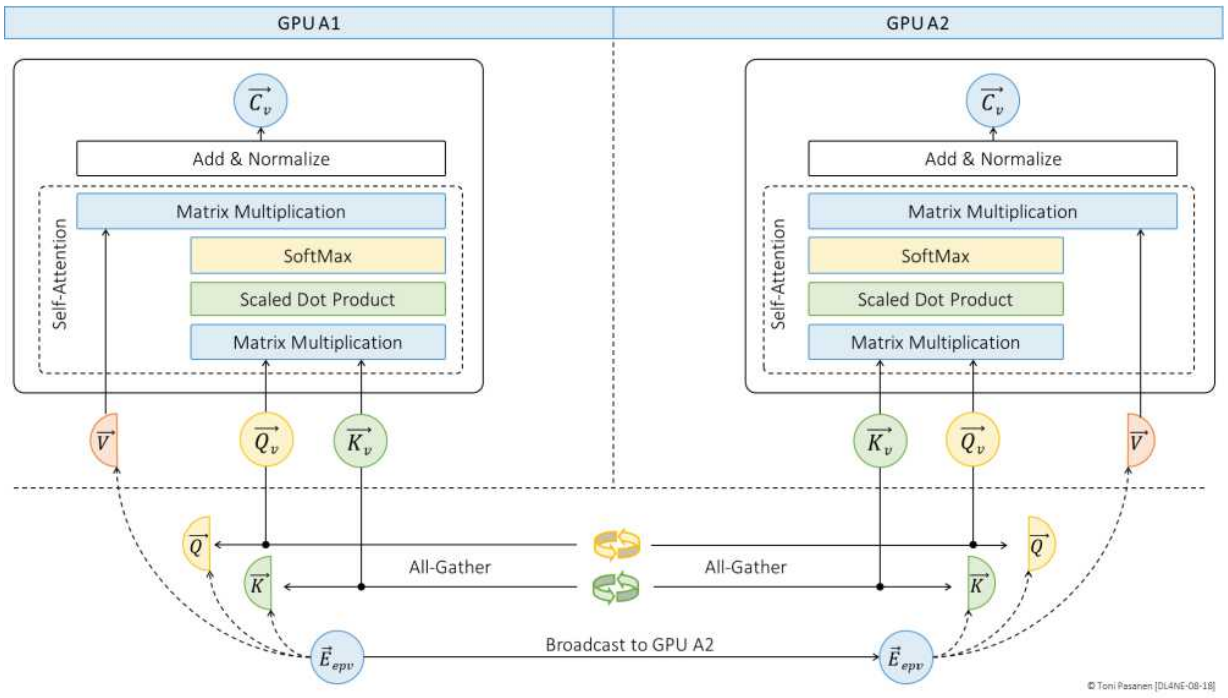


Figure 8-17: *Tensor Parallelism in Self-Attention Layer.*

Feedforward Layer

Figure 8-18 illustrates a Feedforward layer in the decoder module of a transformer. The feedforward network consists of two hidden layers and an output layer. In addition to Tensor Parallelism, we also employ Model Parallelism with Pipeline Parallelism.

The first hidden layer is split between GPU A1 and GPU B1, both located in the same server. The weight matrices for neurons 1–3 reside in GPU A1, while the weight matrices for neurons 4–6 are in GPU B1. The inter-GPU communication between GPU A1 and GPU B1 occurs over NVLinks, which I refer to as the High-speed Domain (HsD).

The second hidden layer is distributed across GPU A2 and GPU B2 within the same server. GPU A2 holds the weight matrices for neurons 1–2, while GPU B2 contains the weight matrices for neurons 3–4. The inter-GPU connection between GPU A2 and GPU B2 also utilizes NVLinks.

The output layer is divided between GPU A3 and GPU B3, both residing in the same server. The weight matrix for neuron 1 is stored in GPU A3, while the weight matrix for neuron 2 is in GPU B3. Inter-GPU communication occurs over NVLinks.

Additionally, GPU A1, GPU A2, and GPU A3 are interconnected via Rail Switch-1 across the Backend Network. Similarly, GPU B1, GPU B2, and GPU B3 are connected via Rail Switch-2 across the Backend Network.

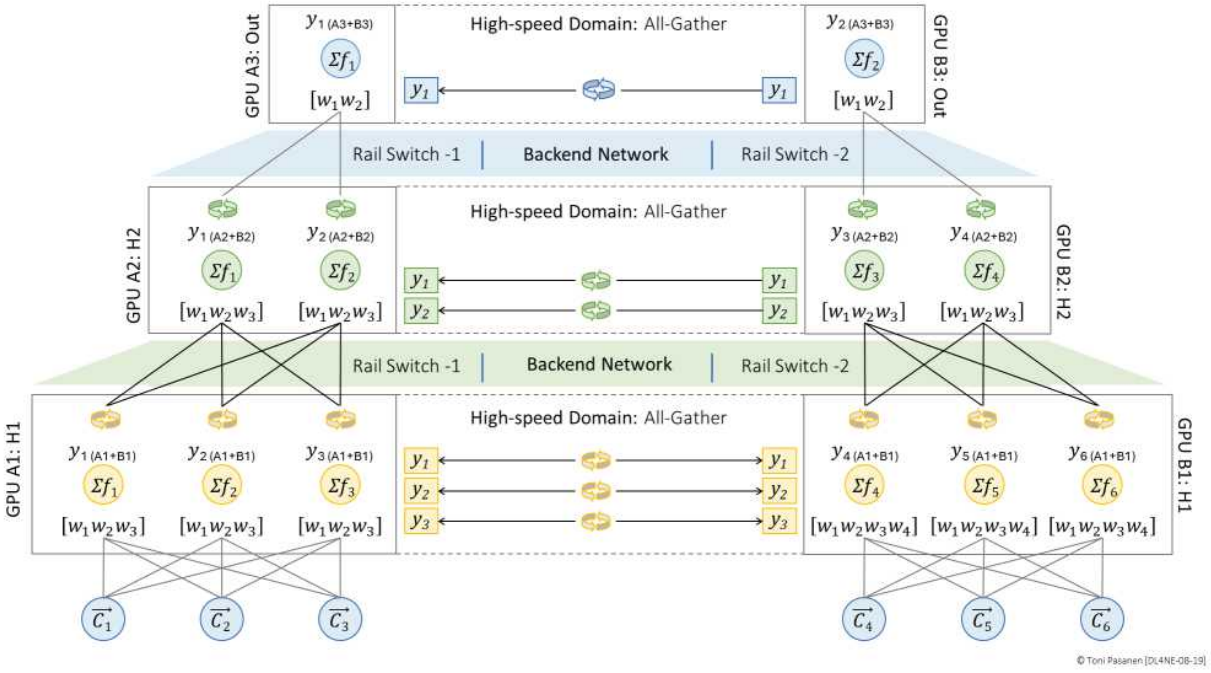


Figure 8-18: *Tensor, Model and Pipeline Parallelism in Feedforward Layer.*

Backpropagation

Forward pass

First Hidden Layer (H1): The input to H1, the output of the Self-Attention block after the Add & Norm step (context vectors), is shared with GPU A1 and GPU B1. Each GPU then performs its local matrix multiplication. After these local computations are complete, the partial outputs are synchronized between GPU A1 and GPU B1 using an All-Gather operation. This synchronization ensures that the complete H1 output ($ynA1+B1$) is

calculated before it is passed to the next stage. Because GPU A1 and GPU B1 reside on the same server, the communication occurs over a high-speed domain via NVLink.

In the context of pipeline parallelism, H1 constitutes one pipeline stage. Once its context vector-based output is fully assembled, it is sent to the GPUs responsible for the next layer. Specifically, GPU A1 and GPU B1 first pass the output computed from the first context vector (C1), and then the GPUs process the next context vector. This communication occurs over the backend network. GPU A1, GPU A2, and GPU A3 are all connected to the same rail switch, so the RDMA packets traverse only one switch. The same design applies to GPU B1, GPU B2, and GPU B3. If communication between GPUs connected to different rail switches is required, the rail switches must be interconnected via spine switches.

Second Hidden Layer (H2): The complete output from H1 (obtained after synchronization in the previous stage) is pipelined to GPUs A2 and B2. Each of these GPUs performs its own local matrix multiplication. As before, after the local computations, the partial outputs from GPU A2 and GPU B2 are synchronized via an All-Gather operation, forming the complete H2 output ($ynA2+B2$).

The synchronization and forwarding between hidden layer 2 and output layer, and within an output layer follow the same model as in the previous hidden layers.

This hybrid approach, using tensor parallelism within each stage and pipeline parallelism across stages, helps balance the computational load and memory usage across the six GPUs while minimizing idle time.

Although the focus of this section is on tensor parallelism, pipeline parallelism is also discussed because large language models (LLMs) can process multiple sentences from their vocabulary simultaneously during the training process.

On the other hand, during the inference when answering to our questions, LLMs use autoregressive next-word prediction. In this process, the final SoftMax layer of the Transformer calculates the probabilities over the vocabulary to predict the next token. This predicted token is then converted into a word and mapped to a new token. The lookup process assigns the token a positional embedding vector, which is used to compute the Query, Key, and Value vectors that feed into the

Transformer's self-attention layer. Consequently, pipeline parallelism is not required during the inference phase.

Backward pass

The error propagates backward from the Feedforward Neural Network (FFNN) layer to the Self-Attention layer. The backpropagation process in a Transformer follows a sequential order, meaning the error from the output propagates first to the FFNN layer, and from there, it continues backward to the Self-Attention mechanism.

The process begins at the output layer, where the error is computed using the SoftMax function and cross-entropy loss. This error is then backpropagated through the FFNN layer, where gradients for the weight matrices are computed. Since the FFNN weights are split across multiple GPUs in Tensor Parallelism, each GPU computes its local gradient. An AllReduce operation is then performed to synchronize these gradients across GPUs, ensuring that all GPUs have the correct weight updates before proceeding.

Once the gradients for the FFNN weights are synchronized, the error propagates back to the Self-Attention layer. Here, gradients for the Query (Q), Key (K), and Value (V) matrices are computed. Since these matrices were split across GPUs during the forward pass, the missing Q and K fragments must be gathered before calculating gradients. An All-Gather operation is used to collect Q and K values across GPUs. Once each GPU

has a complete Q and K matrix, it computes the required gradients locally. After the local gradient computation, an All-Reduce operation is performed to ensure all GPUs have the synchronized gradients before updating the weights.

After both layers complete their gradient computations and synchronizations, the optimizer updates the weights, and the next iteration begins. The key communication phases include All-Gather for assembling required Q and K values before gradient computation and All-Reduce for synchronizing gradients before weight updates.

REFERENCES

[1] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., & Ng, A. Y. (2012, January 1). Large Scale Distributed Deep Networks. Advances in Neural Information Processing Systems, 25, 1223–1231.

https://papers.nips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f6 6102863-Paper.pdf

[2] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., & Chen, Z. (2019, June 10). GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. Advances in Neural Information Processing Systems, 32. https://arxiv.org/pdf/1811.06965.pdf

[3] Narayanan, D., Santhanam, K., Zhao, Y., Shoeybi, M., LeGresley, P., Patwary, M., Thakur, A., Prabhumoye, S., Hall, J., Houston, M., & Zaharia, M. (2021, March 5). Efficient Large-Scale Language Model Training on GPU Clusters using Megatron-LM. Proceedings of the 5th MLSys Conference.

https://arxiv.org/pdf/2104.04473.pdf

[4] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019, December 8). Megatron-LM: Training MultiBillion Parameter Language Models Using Model Parallelism. arXiv preprint.

https://arxiv.org/pdf/1909.08053.pdf

CHAPTER 9: RDMA BASICS

INTRODUCTION

Remote Direct Memory Access (RDMA) architecture enables efficient data transfer between Compute Nodes (CN) in a High-Performance Computing (HPC) environment. RDMA over Converged Ethernet version 2 (RoCEv2) utilizes a routed IP Fabric as a transport network for RDMA messages. Due to the nature of RDMA packet flow, the transport network must provide lossless, low-latency packet transmission. The RoCEv2 solution uses UDP in the transport layer, which does not handle packet losses caused by network congestion (buffer overflow on switches or on a receiving Compute Node). To avoid buffer overflow issues, Priority Flow Control (PFC) and Explicit Congestion Notification (ECN) are used as signaling mechanisms to react to buffer threshold violations by requesting a lower packet transfer rate (Chapter 11 describes these in detail).

Before moving to RDMA processes, let's take a brief look at our example Compute Nodes. Figure 9-1 illustrates our example Compute Nodes (CN). Both Client and Server CNs are equipped with one Graphical Processing Unit (GPU). The GPU has a RDMA capable Network Interface Card (RNIC) with one interface. Additionally, the GPU has Device Memory Units to which it has a direct connection, bypassing the CPU. In real life, a CN may have several GPUs, each with multiple memory units. Intra-GPU communication within the CN happens over high-speed

NVLink. The connection to remote CNs occurs over the NIC, which has at least one highspeed uplink port/interface.

Figure 9-1 also shows the basic idea of a stacked Fine-Grained 3D DRAM (FG-DRAM) solution. In our example, there are four vertically interconnected DRAM dies, each divided into eight Banks. Each Bank contains four memory arrays, each consisting of rows and columns that

contain memory units (transistors whose charge indicates whether a bit is set to 1 or 0). FG-DRAM enables cross-DRAM grouping into Ranks, increasing memory capacity and bandwidth.

The upcoming sections introduce the required processes and operations when the Client Compute Node wants to write data from its device memory to the Server Compute Node's device memory. I will discuss the design models and requirements for lossless IP Fabric in later chapters.

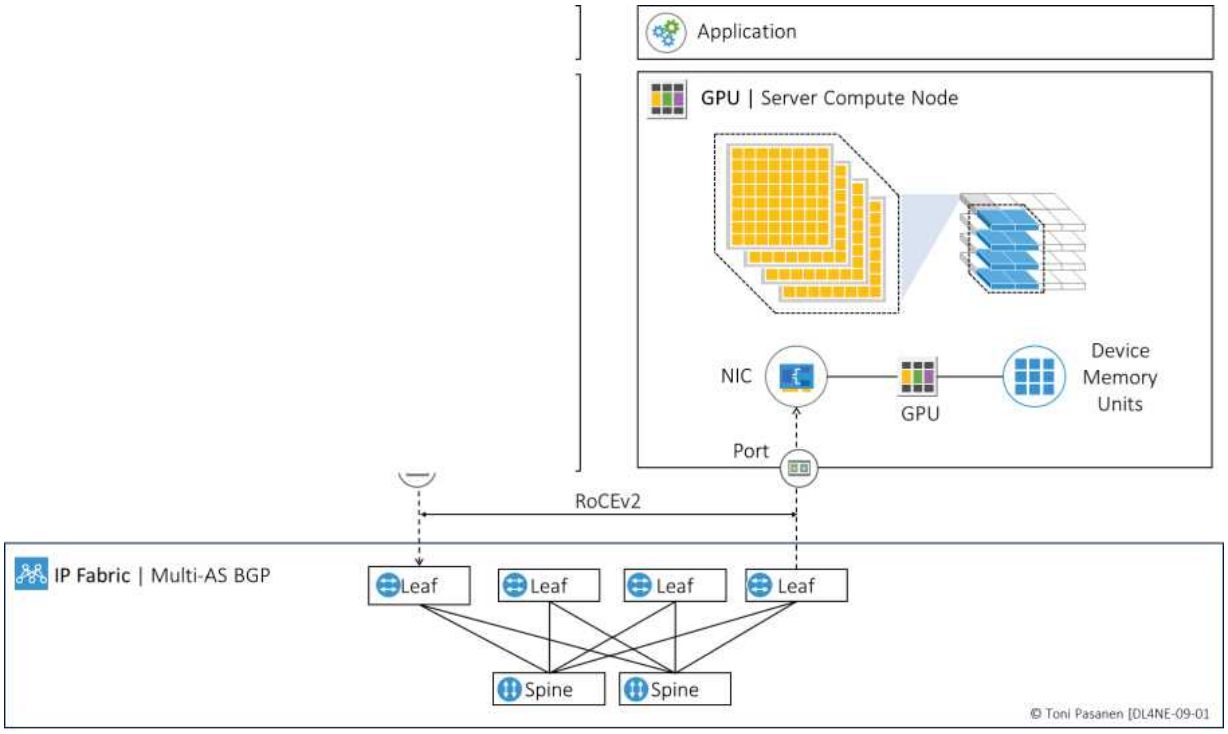
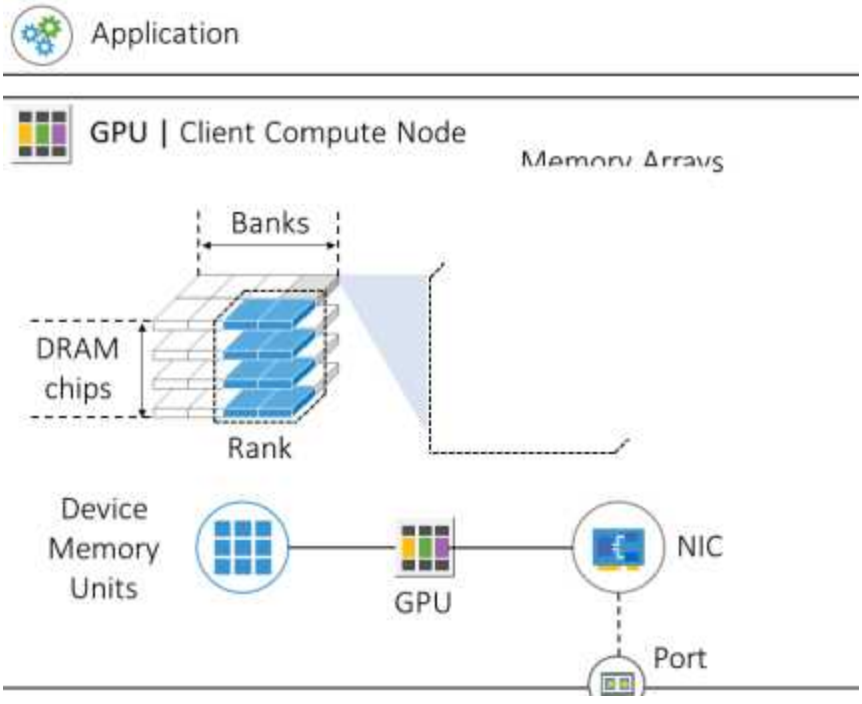
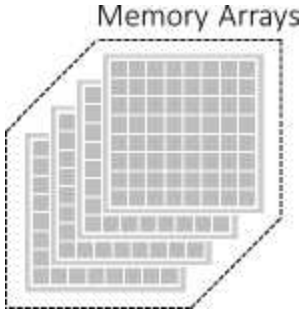


Figure 9-1: *Fine-Grained DRAM High-Level Architecture.*




Memory Arrays

AN OVERVIEW OF RDMA PROCESSES

The focus of the following sections is to provide an overview of the processes involved when our example application writes data from the Client Compute Node's (CCN) device memory to the Server Compute Node's (SCN) device memory. This section is divided into four paragraphs:

> a) memory allocation and registration, b) creation of queue pairs, c) connection initiation, and d) write operations from the perspectives of both the CCN and SCN.

Memory Allocation and Registration

First, we allocate a Protection Domain (PD). You can think of the PD as a tenant in IP networking. It enables the creation of a dedicated, private environment for your objects, like a Virtual Routing and Forwarding (VRF) instance in traditional networking, where the "objects" are IP addresses and routing tables. After allocating the PD, we allocate a memory block from the physical device memory and register it. During the memory registration process, we define the size of the memory block and set its access rights. In our example, we have set the access rights for registered memory in the CCN to Local Read and in the SCN

to Remote Write. Next, we associate the registered device memory space with the PD.

Note that the allocated physical memory may not be contiguous; therefore, the registration process creates a virtual, contiguous memory block. As a result of these processes, we receive a local memory access key, or L_Key. When we register memory for an RDMA write operation and assign it Remote Write access, we receive a remote memory key, or R_Key. This is the case with the SCN. The R_Key is sent to the CCN over a management connection (which is outside the scope of this chapter). At this phase, both nodes have registered device memory associated with the PD. Additionally, L_Keys and R_Keys are generated.

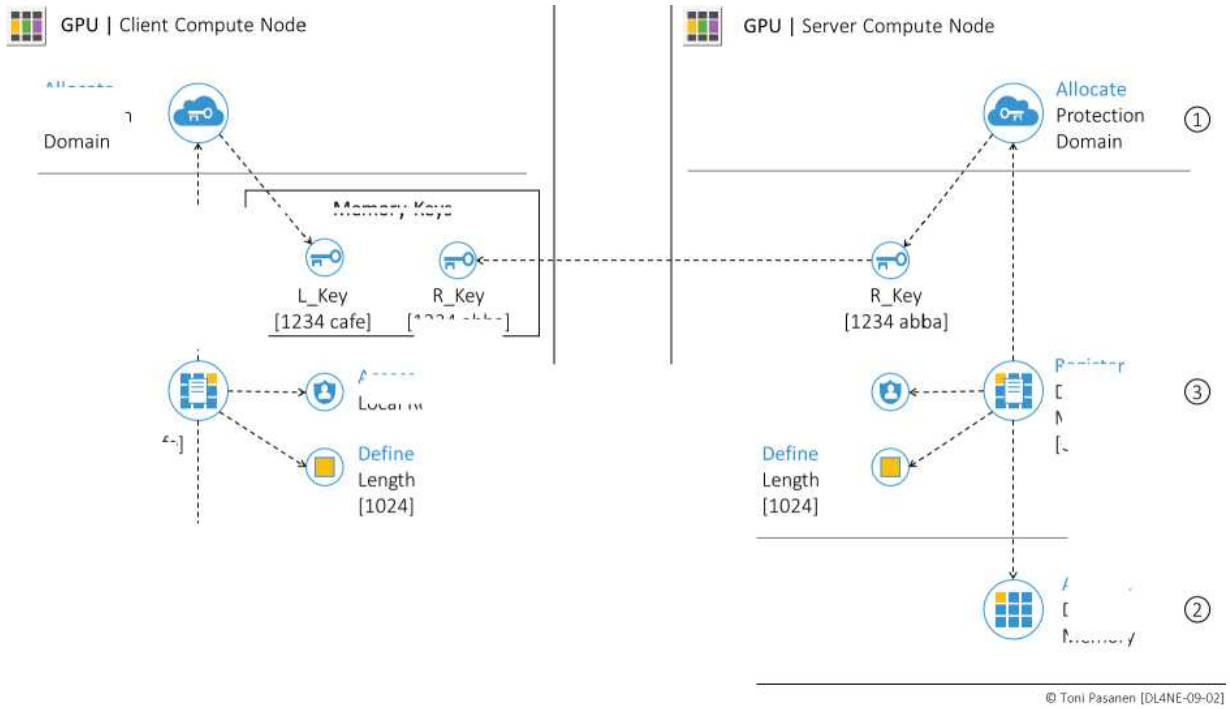


Figure 9-2: *Memory Allocation and Registration.*

Create Queue Pairs

A Work Queue (WQ) is a bi-directional virtual communication channel between the RNIC and the device memory. The WQ consists of two queues: the send queue (for RDMA send and write operations) and the receive queue (for receive operations). A Queue Pair (QP) is composed of these two queues. A Completion Queue (CQ) is used to notify the application of the completion status of an RDMA operation. Each QP is assigned a Service Type, which defines the connection's service level (Reliable or Unreliable) and type (Connection: point-to-point or Datagram: point-to-multipoint). In our example, we are using a Reliable Connection (RC).

When creating a Queue Pair, we bind it with the same PD to which our registered virtual memory block is associated. We also bind the send and receive queues to either the same or different completion queues. Next, we set the service type for the QP. During the QP creation process, we also define the maximum number of send and receive Work Requests and their maximum message size.

To establish a communication channel between Compute Nodes, the port on the NIC of the CCN and SCN must belong to the same partition. Each port of the NICs acts as an Endpoint in the RDMA domain. Every port has a Partition Key (P_Key) Table

with at least one P_Key. After creating a Queue Pair, we query a P_Key from the specific port, set the QP state to INIT (initialize), and set the P_Key value. The CCN sends the P_Key to the SCN within the Connection Request during the connection initiation process. The CCN includes the P_Key in every RDMA message, and the receiving node verifies that the P_Key in the datagram matches the target QP's P_Key.

You can think of the P_Key as a virtual connection identifier for Queue Pairs, similar to how the VXLAN Network Identifier (VNI) in a VXLAN header identifies a VXLAN segment. In our example, the QP on the CCN is identified as 0x12345678, and the P_Key associated with it is 0x8012.

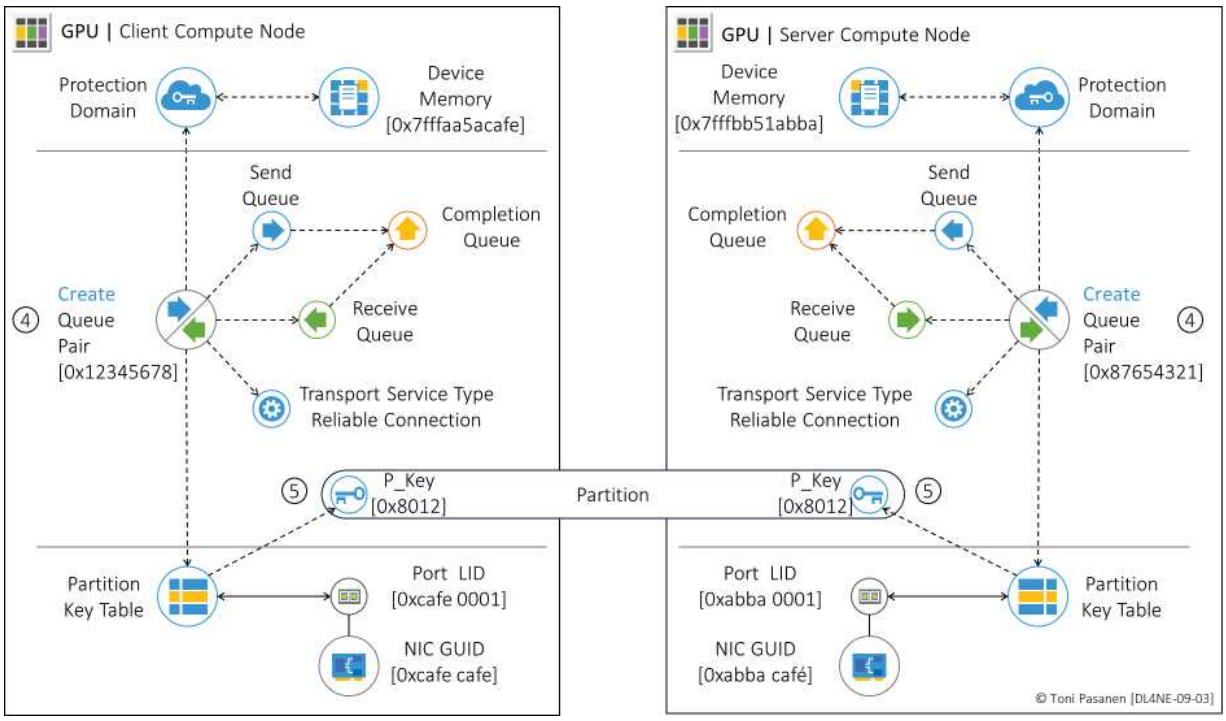


Figure 9-3: *Create Queue Pairs*.

RDMA Connection Initiation

At this phase, the application on the CCN starts the connection initialization by sending a Request for Communication (REQ) message to the application on the SCN. The REQ message includes the Local Communication Identifier (LID) and the Global Unique Identifier for the Channel Adapter (Local CA GUID). The Local CA GUID identifies the NIC, while the Local Communication ID identifies the port on the NIC. The REQ message also carries the Local QP number (0x12345678), QP service type (Reliable Connection), starting Packet Sequence Number (PSN: 0x000abc), P_Key value (0x8012), and payload size (1024).

The Reply message from the SCN describes the local and remote Communication IDs, QP number, and PSN. The CCN responds to the Reply message with a Ready to Use (RTU) message. During the connection initialization process, the QP state transitions from INIT to Ready to Send and Ready to Receive states. After the connection is initiated, the application on the CCN can start the RDMA Write process.

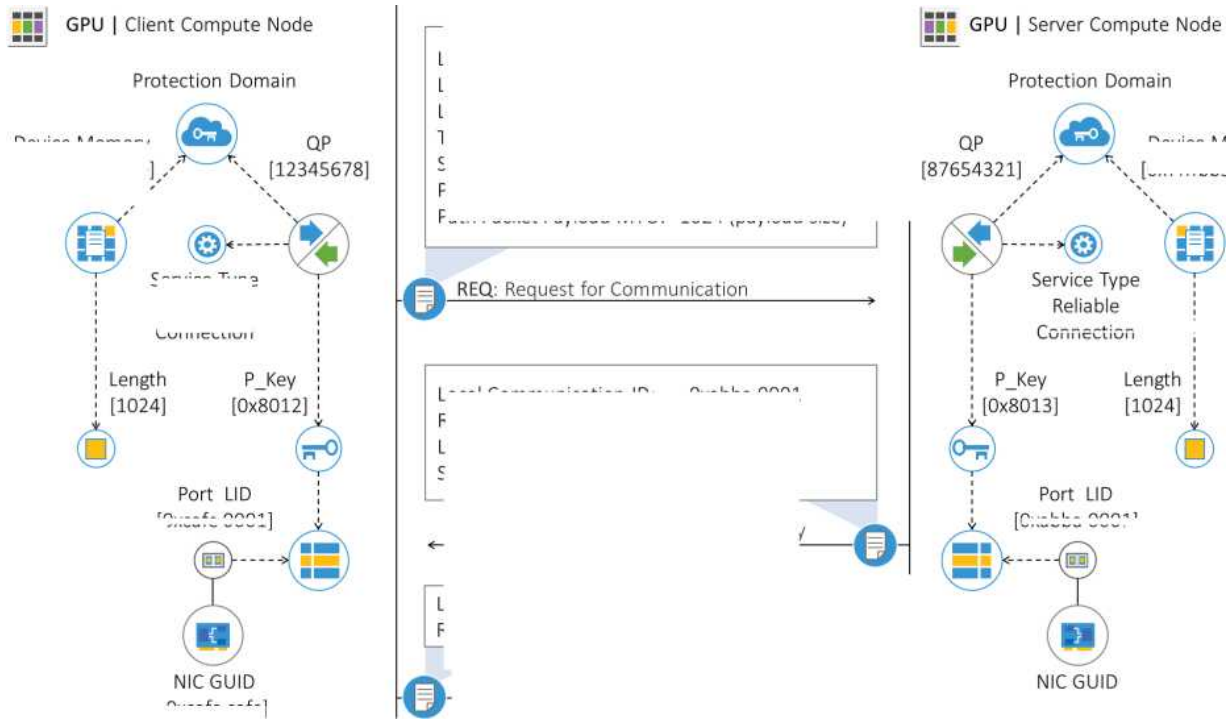


Figure 9-4: *RDMA Connection Initiation.*

Work Request Message

After successfully initiating the connection, the application on the CCN can begin the RDMA Write operation. It creates a Work Request (WR) and posts it to the assigned QP's send queue as a Work Request Entity (WRE). The WRE contains the following information:

- **Work Request Identifier:** Identifies the WR and serves as a pointer in the completion queue to signal the application when the WR has been processed.

- **OpCode:** Specifies the type of operation, such as RDMA Write in our example.

▪ **Local Buffer Address and Length:** Describes the location in local device memory from which data is written to SCN memory, along with the length of the data.

▪ **Local Memory Key (L_Key):** Used for accessing the local memory buffer.

▪ **Send Flag:** Indicates that successful processing of the WR should be signaled to the application through the Completion Queue.

▪ **Remote Buffer Address:** Specifies the target memory location on the SCN.

▪ **Remote Key (R_Key):** Used for accessing the remote memory buffer on the SCN. The R_Key is received over management connection from the SCN.

The NIC retrieves the Work Request from the send queue. Based on the WR information, it constructs an InfiniBand Base Transport Header (IB BTH), which includes the P_Key and Destination QP identifier obtained during

the connection initiation process. Because of the Reliable Connection service type, the Ack Required value is set to Yes.

The RDMA Write operation requires an RDMA Extended Transport Header (RETH), which details the destination memory buffer, R_Key, and data length. The IB BTH and RETH headers are encapsulated within Ethernet/IP/UDP headers. The destination port 4791 in the UDP header indicates that the next header is IB BTH.

Wrapped inside Eth/IP/UDP/IB BTH/RETH headers the data is forwarded towards the SCN.

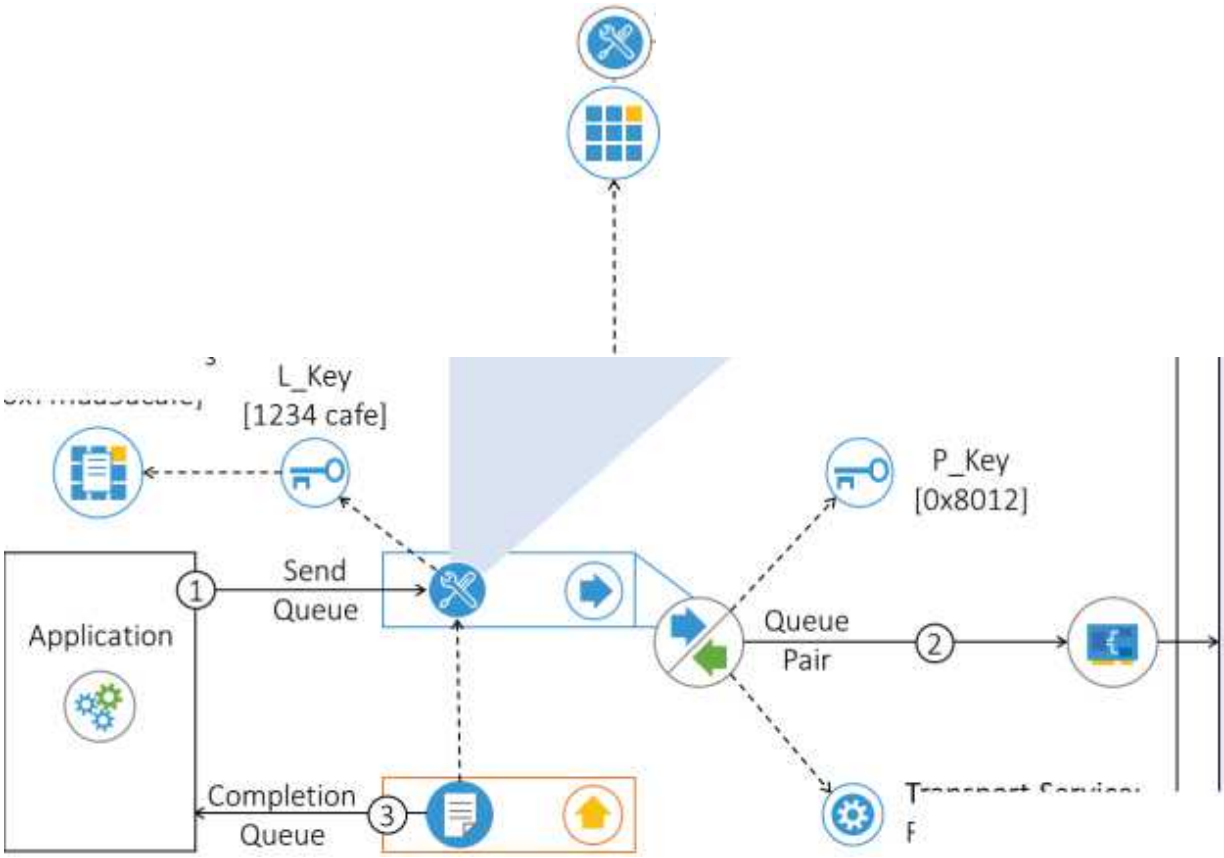


Figure 9-5: *Work Request Message – Step 1.*

When the SCN receives the RDMA Write message, it checks the received P_Key and assigns the ingress port. Additionally, it validates the R_Key to ensure it matches what was published to the CCN over the management connection. After these validations, the NIC translates the virtual device memory address to physical memory access and sends the RDMA Write information to the QP's Receive Queue. Finally, once the RDMA write operation is completed, the application is notified via the Completion Queue that the job is done.

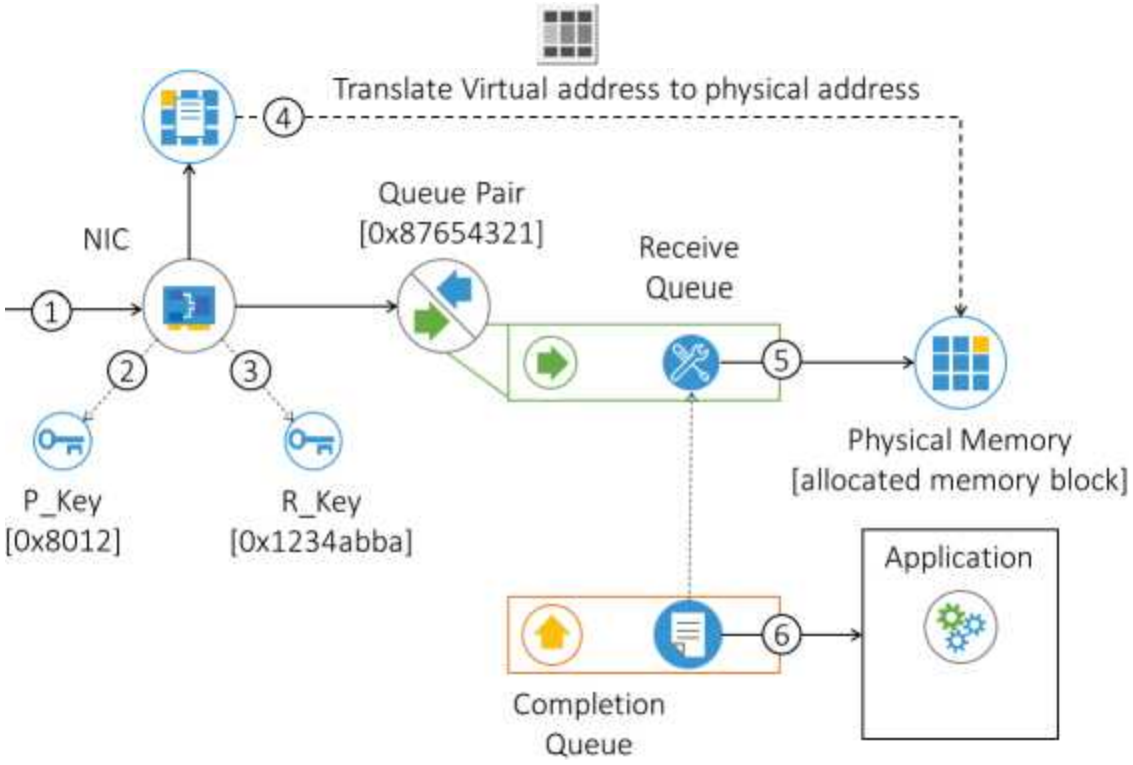


Figure 9-6: *Work Request Message – Step 2.*

REFERENCES

[1] InfiniBand YM Architecture Specification Volume 1, Release 1.7, July

11, 2023, Final

CHAPTER 10: CHALLENGES IN AI FABRIC DESIGN

INTRODUCTION

Figure 10-1 illustrates a simple distributed GPU cluster consisting of three GPU hosts. Each host has two GPUs and a Network Interface Card (NIC) with two interfaces. Intra-host GPU communication uses high-speed NVLink interfaces, while inter-host communication takes place via NICs over slower PCIe buses.

GPU-0 on each host is connected to Rail Switch A through interface E1. GPU-1 uses interface E2 and connects to Rail Switch B. In this setup, interhost communication between GPUs connected to the same rail passes through a single switch. However, communication between GPUs on different rails goes over three hops Rail–Spine–Rail switches.

In Figure 10-1, we use a data parallelization strategy where a training dataset is split into six micro-batches, which are distributed across the GPUs. All GPUs use the shared feedforward neural network model and compute local model outputs. Next, each GPU calculates the model error and begins the backward pass to compute neuron-based gradients. These gradients indicate how much, and in which direction, the weight parameters should be adjusted to improve the training result (see Chapter 2 for details).

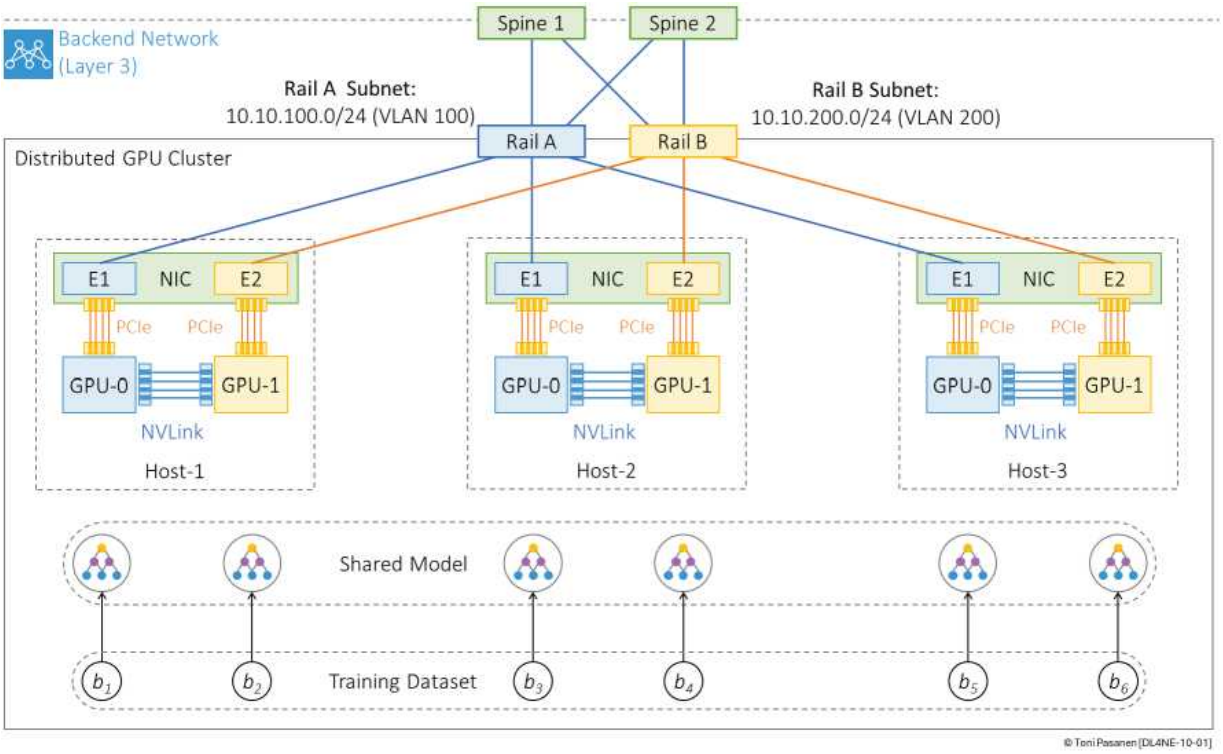


Figure 10-1: *Rail-Optimized Topology.*

EGRESS INTERFACE CONGESTIONS

After computing all gradients, each GPU stores the results in a local memory buffer and starts a communication phase where computed gradients are shared with other GPUs. During this process, the data (gradients) is being sent from one GPU and written to another GPU's memory (RDMA Write operation). RDMA is explained in detail in Chapter 9.

Once all gradients have been received, each GPU averages the results (AllReduce) and broadcasts the aggregated gradients to the other GPUs. This ensures that all GPUs update their model parameters (weights) using the same gradient values. The Backward pass process and gradient calculation are explained in Chapter 2.

Figure 10-2 illustrates the traffic generated during gradient synchronization from the perspective of GPU-0 on Host-2. Gradients from the local host’s GPU-1 are received via the high-speed NVLink interface,

while gradients from GPUs in other hosts are transmitted over the backend switching fabric. In this example, all hosts are connected to Rail Switches using 200 Gbps fiber links. Since GPUs can communicate at line rate, gradient synchronization results in up to 800 Gbps of egress traffic toward interface E1 on Host-2, via Rail Switch A. This may cause congestion, and packet

drops if the egress buffer on Rail Switch A or the ingress buffer on interface E1 is not deep enough to accommodate the queued packets.

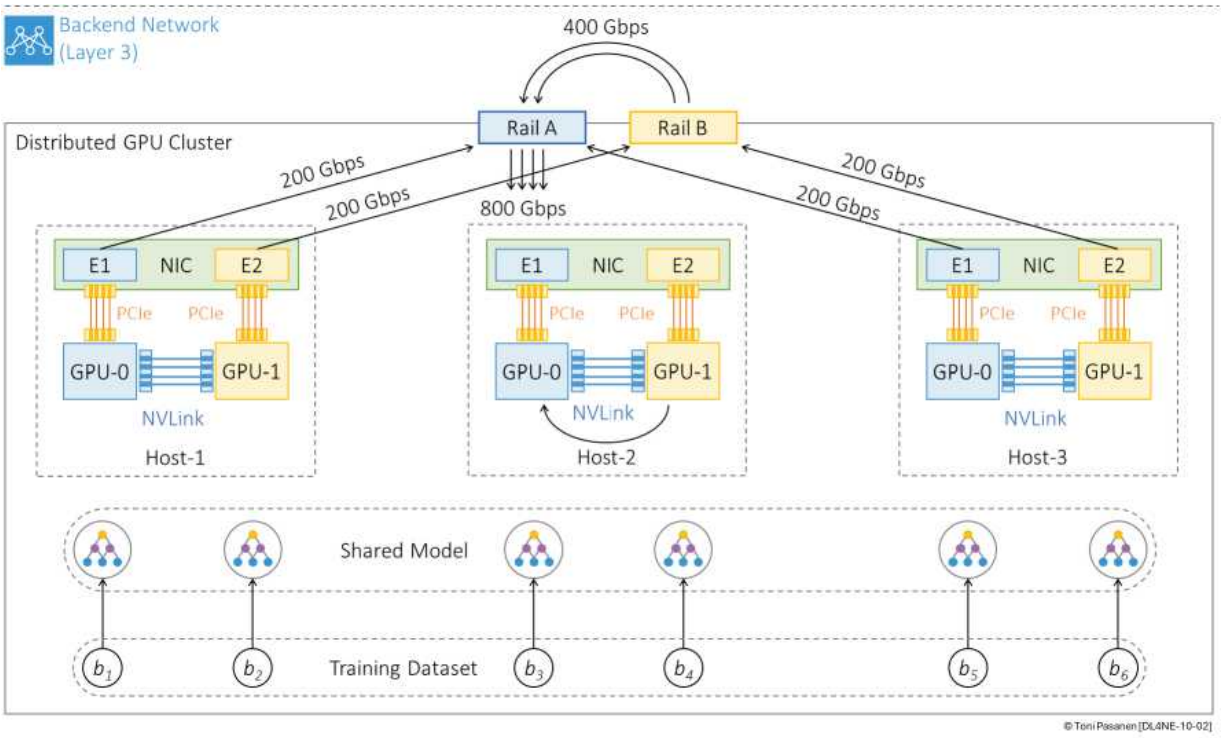


Figure 10-2: *Congestion During Backward Pass.*

SINGLE POINT OF FAILURE

The training process of a neural network is a long-running, iterative task where GPUs must communicate with each other. The frequency and pattern of this communication depend on the chosen parallelization strategy. For example, in data parallelism, communication occurs during the backward pass, where GPUs synchronize gradients. In contrast, model parallelism and pipeline parallelism involve communication even during the forward pass, as one GPU sends activation results to the next GPU holding the subsequent layer. It is important to understand that communication issues affecting even a single GPU can delay or interrupt

the entire training process. This makes the AI fabric significantly more sensitive to single points of failure compared to traditional data center fabrics.

Figure 10-3 highlights several single points of failure that may occur in real-world environments. A host connection can become degraded or completely fail due to issues in the host, NIC, rail switch, transceiver, or connecting cable. Any of these failures can isolate a GPU. While this might not seem serious in large clusters with thousands of GPUs, as discussed in the previous section, even one isolated or failed GPU can halt the training process.

Problems with interfaces, transceivers, or cables in inter-switch links can cause congestion and delays. Similar issues arise if a spine switch is malfunctioning. These types of failures typically affect inter-rail traffic but not intra-rail communication. A failure in a rail switch can isolate all GPUs connected to that rail, creating a critical point of failure for a subset of the GPU cluster.

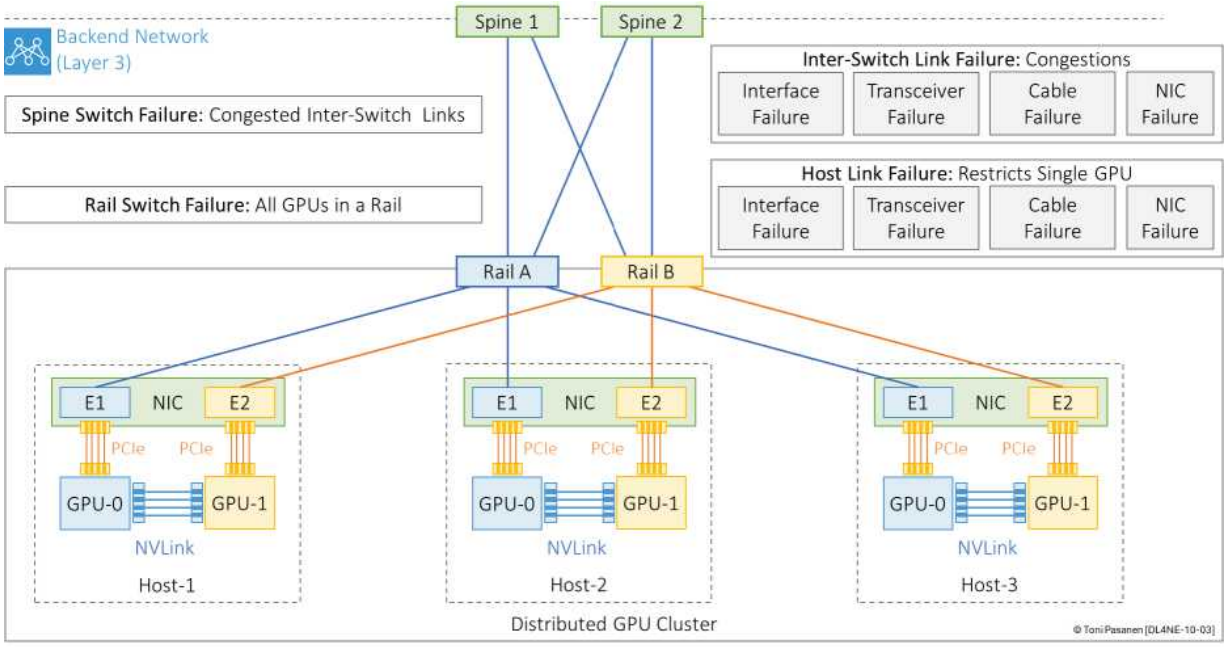


Figure 10-3: *Single-Point Failures.*

HEAD-OF-LINE BLOCKING

In this example GPU clusters, NCCL (NVIDIA Collective Communications Library) has built a topology where gradients are first sent from GPU-0 to GPU-1 over NVLink, and then forwarded from GPU-1 to other GPU-1s via Rail switch B.

However, this setup may lead to head-of-line blocking. This happens when GPU-1 is already busy sending its own gradients to the other GPUs, and now it also needs to forward GPU-0's gradients. Since the PCIe and NIC bandwidth is limited, GPU-0's traffic may need to wait in line behind GPU-1's traffic. This queueing delay is called head-of-line blocking, and it can slow down the whole training process. The problem is more likely to happen when many GPUs rely on a single GPU or NIC for forwarding traffic to another rail. Even if only one GPU is overloaded, it can cause delays for others too.

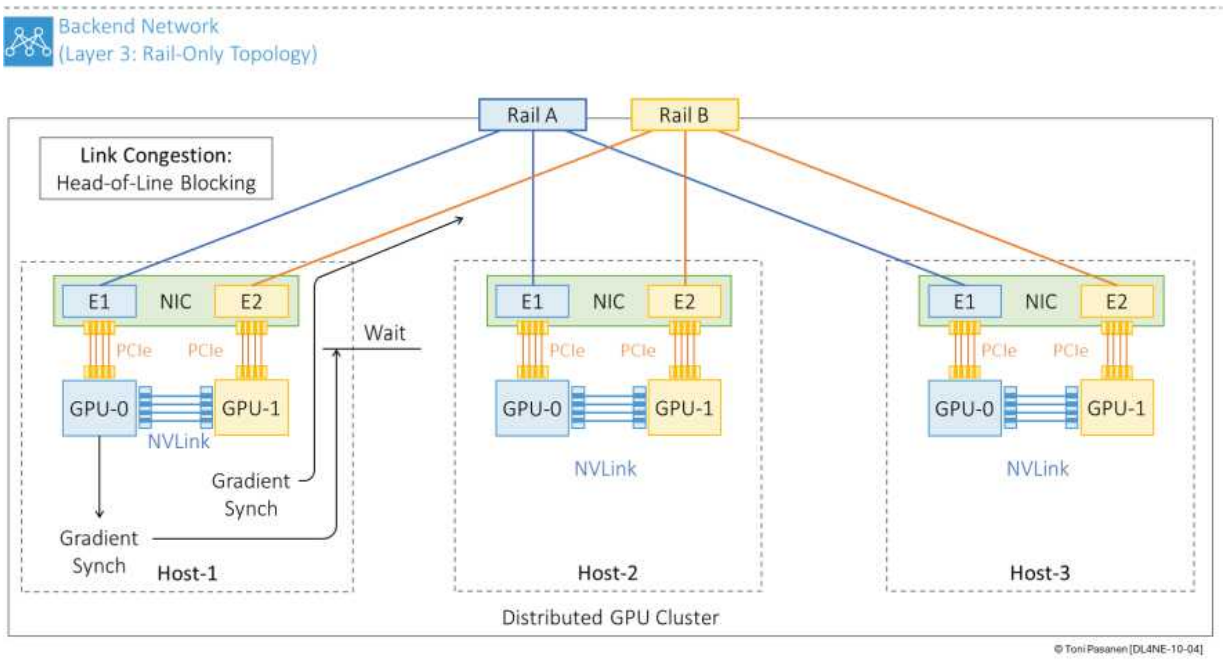


Figure 10-4: *Head-of-Line Blocking.*

HASH-POLARIZATION WITH ECMP

First, when two GPUs open a Queue Pair (QP) between each other, all gradient synchronization traffic is typically sent over that QP. From the network point of view, this looks like one large flow between the GPUs. In deep learning training, gradient data can be hundreds of megabytes or even gigabytes, depending on the model size. So, when it is sent over one QP, the network sees it as a single high-bandwidth flow. This kind of traffic is often called an elephant flow, because it can take a big share of the link bandwidth. This becomes a problem when multiple large flows are hashed to the same uplink or spine port. If that happens, one link can get overloaded while others remain underused. This is one of the reasons we see hash polarization and head-of-line blocking in AI clusters. Hash polarization is a condition where the load-balancing hash algorithm used in ECMP (Equal-Cost Multi-Path) forwarding results in uneven distribution of traffic across multiple available paths.

For example, in Figure 10-5, GPU-0 in Host-1 and GPU-0 in Host-2 both send traffic to GPU-1 at a rate of 200 Gbps. The ECMP hash function in Rail Switch A selects the link to Spine 1 for both flows. This leads to a situation where one spine link carries 400 Gbps of traffic, while the other remains idle. This is a serious problem in AI clusters because training jobs generate large volumes of east-west traffic between GPUs, often at line rate.

When traffic is unevenly distributed due to hash polarization, some links become congested while others are idle. This causes packet delays and retransmissions, which can slow down gradient synchronization and reduce the overall training speed. In large-scale clusters, even a small imbalance can have a noticeable impact on job completion time and resource efficiency.

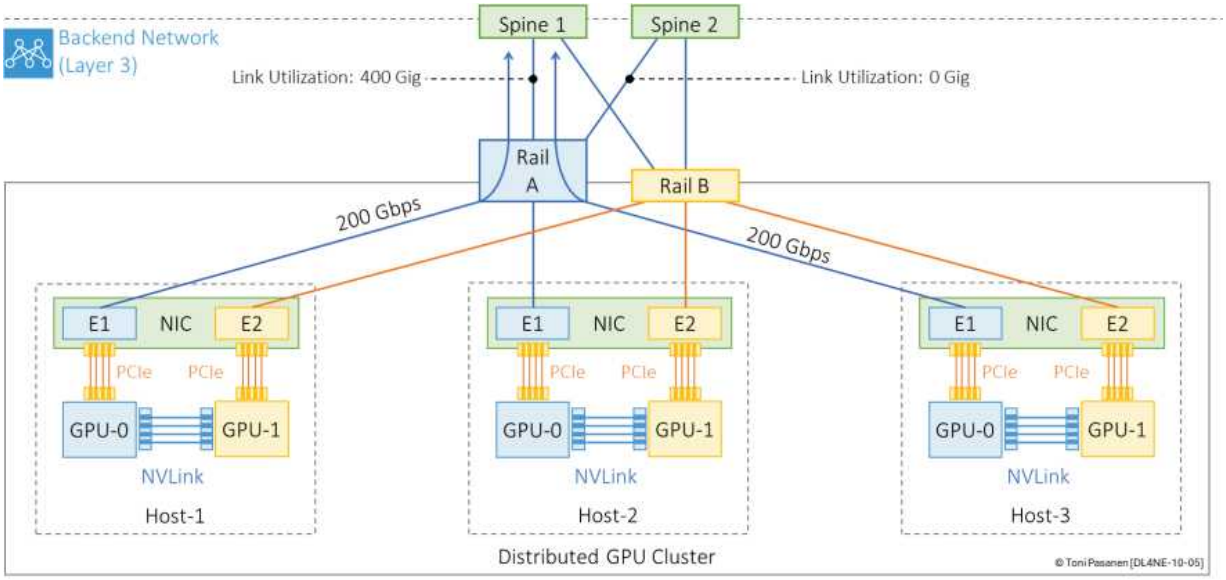


Figure 10-5: *ECMP Hash-Polarization.*

In the previous sections, we explored some of the key challenges that can impact performance and reliability in GPU-based AI clusters, such as link congestion, single points of failure, head-of-line blocking, and hash polarization in ECMP routing.

The rest of this book focuses on how these problems can be mitigated or even fully avoided. We will look at design choices, transport optimizations, network-aware scheduling, and

alternative topologies that help improve the robustness and efficiency of the AI fabric.

REFERENCES

[1] Yunhong Xu, Keqiang He, Rui Wang, Minlan Yu, Nick Duffield, Hassan Wassel, Shidong Zhang, Leon Poutievski, Junlan Zhou, Amin Vahdat, Hashing Design in Modern Networks: Challenges and Mitigation Techniques. Texas A&M University, Harvard University, Google.

https://www.usenix.org/system/files/atc22_slides_xu.pdf

[2] Head-of-line blocking, Wikipedia, https://en.wikipedia.org/wiki/Head-of-line_blocking

CHAPTER 11: CONGESTION AVOIDANCE

As explained in the preceding chapter, "*Egress Interface Congestions*," both the Rail switch links to GPU servers and the inter-switch links can become congested during gradient synchronization. It is essential to implement congestion control mechanisms specifically designed for RDMA workloads in AI fabric back-end networks because congestion slows down the learning process and even a single packet loss may restart the whole training process.

This section begins by introducing Explicit Congestion Notification (ECN) and Priority-based Flow Control (PFC), two foundational technologies used in modern lossless Ethernet networks. ECN allows switches to mark packets, rather than dropping them, when congestion is detected, enabling endpoints to react proactively. PFC, on the other hand, offers per-priority flow control, which can pause selected traffic classes while allowing others to continue flowing.

Finally, we describe how Datacenter Quantized Congestion Notification (DCQCN) combines ECN and PFC to deliver a scalable and lossless transport mechanism for RoCEv2 traffic in AI clusters.

GPU-TO-GPU RDMA WRITE WITHOUT CONGESTION

The figure 11-1 illustrates a standard Remote Direct Memory Access (RDMA) Write operation between two GPUs. This example demonstrates how GPU-0 on Host-1 transfers local gradients (∇_1 and ∇_2) from memory to GPU-0 on Host-2. Both GPUs use RDMA-capable NICs connected to Rail Switch A via 200 Gbps uplinks.

The RDMA Write operation proceeds through the following seven steps:

1. To initiate the data transfer, GPU-0 on Host-1 submits a work request to its RDMA NIC over the PCIe bus over the pre-established Queue Pair 0x123456.

2. The RDMA NIC encodes the request by inserting the OpCode (RDMA Write) and Queue Pair Number (0x123456) into the InfiniBand Transport Header (IBTH). It wraps the IBTH and RETH (not shown in the figure) headers with Ethernet, IP, UDP, and Ethernet headers. The NIC sets the DSCP value to 24 and the ECN bits to 10 (indicating ECN-capable transport) in the IP header's ToS octet. The DSCP value ensures that the switch can identify and prioritize RoCEv2 traffic. The destination UDP port is set to 4791 (not shown in the figure).

3. Upon receiving the packet on interface Ethernet1/24, the Rail switch classifies the traffic as RoCEv2 based on the DSCP value of 24.

4. The switch maps DSCP 24 to QoS-Group 3.

5. QoS Group 3 uses egress priority queue 3, which is configured with bandwidth allocation and congestion avoidance parameters (WRED Min, WRED Max, and Drop thresholds) optimized for RDMA traffic.

6. The packet count on queue 3 does not exceed the WRED minimum threshold, so packets are forwarded without modification.

7. The RDMA NIC on Host-2 receives the packet, strips off the Ethernet, IP, and UDP headers, and processes the RDMA headers. The payload is delivered directly into the memory of GPU-0 on Host-2 without CPU involvement, completing the RDMA Write operation.

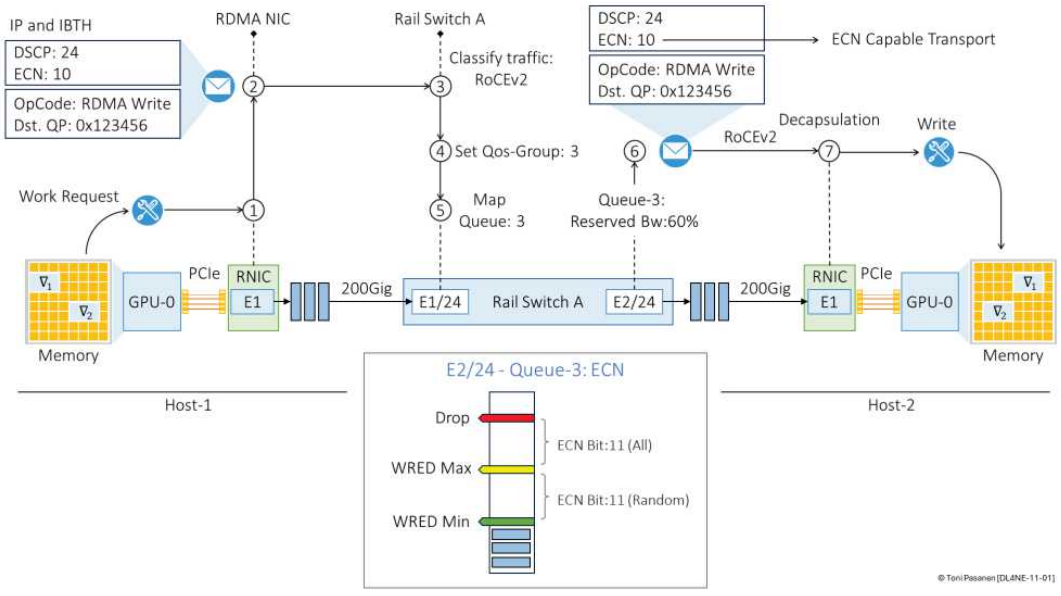


Figure 11-1: *Overview of Remote DMA operation Under Normal Condition.*

EXPLICIT CONGESTION NOTIFICATION -ECN

Because gradient synchronization requires a lossless network service, it is essential to have a proactive congestion detection system that can respond before buffer overflows occur. This system must also include a signalling mechanism that allows the receiver to request the sender to reduce its transmission rate or temporarily pause traffic when necessary.

Data Center Quantized Congestion Notification (DCQCN) is a congestion control scheme designed for RoCEv2 that leverages both Explicit Congestion Notification (ECN) and Priority Flow Control (PFC) for active queue management. This section focuses on ECN.

In IPv4, the last two bits (bits 6 and 7) of the ToS (Type of Service) byte are reserved for ECN marking. With two bits, four ECN codepoints are defined:

00 – Not ECN-Capable Transport (Not-ECT)

01 – ECN-Capable Transport (ECT)

10 – ECN-Capable Transport (ECT)

11 – Congestion Experienced (CE)

Figure 11-2 illustrates how ECN is used to prevent packet drops when congestion occurs on egress interface Ethernet2/24. ECN

operates based on two queue thresholds: WRED Minimum and WRED Maximum. When the queue depth exceeds the WRED Minimum but remains below the WRED Maximum, the switch begins to randomly mark forwarded packets with ECN 11 (Congestion Experienced). If the queue depth exceeds the WRED Maximum, all packets are marked with ECN 11. The Drop Threshold defines the upper limit beyond which packets are no longer marked but instead dropped.

In Figure 11-2, GPU-0 on Host-1 transfers gradient values from its local memory to the memory of GPU-0 on Host-2. Although not shown in the figure for simplicity, other GPUs connected to the same rail switch also participate in the synchronization process with GPU-0 on Host-2. Multiple simultaneous elephant flows towards GPU-0 reach the rail switch, causing egress queue 3 on interface Ethernet2/24 to exceed the WRED Maximum threshold. As a result, the switch begins marking outgoing packets with ECN 11 (step 6), while still forwarding them to the destination GPU.

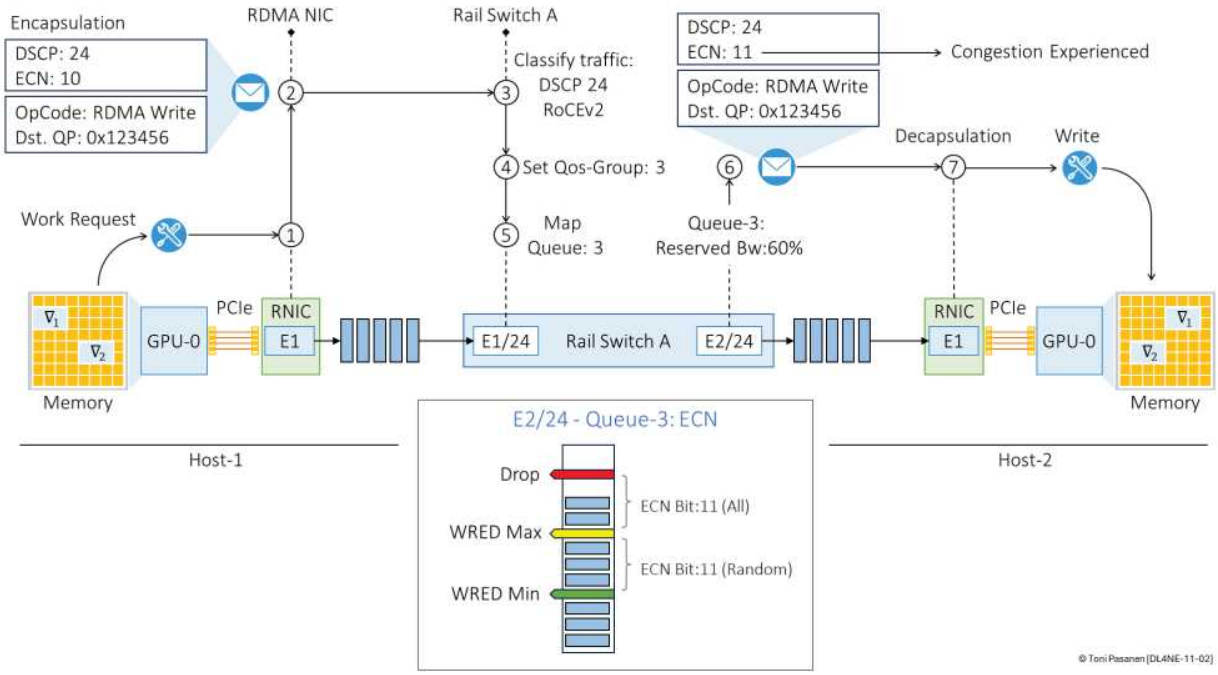


Figure 11-2: *Congested Egress Interface – ECN Congestion Experienced.*

After receiving a data packet marked with ECN 11, the destination RDMA NIC must inform the sender about network congestion. This is accomplished using a Congestion Notification Packet (CNP), which provides feedback at the RDMA transport layer.

1. Upon detecting the ECN 11 in the IP header's ToS bits of an incoming packet, the RDMA NIC on Host-2 generates a CNP message. It sets the OpCode in the IBTH header to 0x81, identifying the message as a CNP. Besides the FECN bit is set to 1, indicating that the NIC experienced congestion. The Queue Pair number used for the original memory copy (e.g., 0x123456) is reused, ensuring the feedback reaches the correct sender-side transport context. In the IP header, the

DSCP field is set to 48, allowing switches to distinguish the CNP from standard RoCEv2 data traffic.

2. When the CNP reaches the Rail switch (interface Ethernet2/24), it is classified based on DSCP 48, which is associated with CNP traffic in the QoS configuration.

3. DSCP 48 maps the packet to QoS group 7, which is reserved for congestion feedback signaling.

4. QoS group 7 is associated with strict-priority egress queue 7, ensuring that CNP packets are forwarded with the highest priority. This guarantees that congestion signals are not delayed behind other types of traffic.

5. The switch forwards the CNP to the originating NIC on Host-1. Because of the strict-priority handling, the feedback arrives quickly even during severe congestion.

6. Upon receiving the CNP, the sender-side RDMA NIC reduces its transmission rate for the affected Queue Pair by increasing inter-packet delay. This is achieved by holding outgoing packets longer in local buffers, effectively reducing traffic injection into the congested fabric.

As transmission rates decrease, the pressure on the egress queue at the Rail switch's interface Ethernet1/14 (connected to Host-2) is gradually relieved. Buffer occupancy falls below the WRED Minimum Threshold, ending ECN marking. Once congestion is fully cleared, the RDMA NIC slowly ramp up its transmission rate. This gradual marking strategy helps prevent

sudden traffic loss and gives the sender time to react by adjusting its sending rate before the buffer overflows.

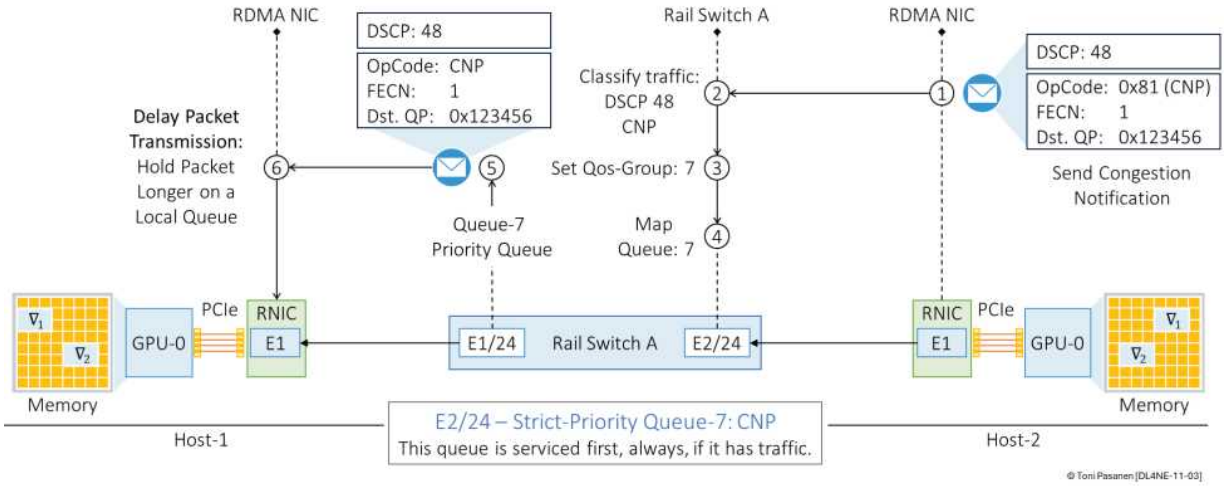


Figure 11-3: *Receiving NIC Generates CNP - Sender NIC Delay Transmit.*

DSCP-BASED PRIORITY FLOW CONTROL (PFC)

Priority Flow Control (PFC) is a mechanism designed to prevent packet loss during network congestion by pausing traffic selectively based on priority levels. While the original IEEE 802.1Qbb standard operates at Layer 2, using the Priority Code Point (PCP) field in Ethernet headers, AI Fabrics rely on Layer 3 forwarding, where traditional Layer 2-based PFC is no longer applicable. To extend lossless behavior across routed (Layer 3) networks, DSCP-based PFC is used.

In DSCP-based PFC, the Differentiated Services Code Point (DSCP) field in the IP header identifies the traffic class or priority. Switches map specific DSCP values to internal traffic classes and queues. If congestion occurs on an ingress interface and a particular priority queue fills beyond a threshold, the switch can send a PFC pause frame back to the sender switch, instructing it to temporarily stop sending traffic of that class—just as in Layer 2 PFC, but now triggered based on Layer 3 classifications.

This behavior differs from Explicit Congestion Notification (ECN), which operates at Layer 3 as well but signals congestion by marking packets instead of stopping traffic. ECN acts on the egress port, informing the receiver to notify the sender to reduce the transmission rate over time. In

contrast, PFC acts immediately at the ingress port, pausing traffic flow in real time to avoid buffer overflows and packet drops.

PFC relies on two thresholds to control flow: xOFF and xON. The xOFF threshold defines the point at which the switch generates a pause frame when a priority queue reaches a congested state. Once triggered, upstream devices halt transmission of that traffic class. The switch continuously monitors its buffer occupancy, and when the level drops below the xON threshold, it sends a PFC frame with a Quanta value of 0 for the affected priority. This signals the upstream device that it can resume transmission for that specific priority queue.

A key requirement for PFC to function correctly is the provisioning of buffer headroom. The switch must reserve enough buffer space per priority class to accommodate in-flight traffic while the pause frame propagates to the sender and takes effect.

DSCP-based PFC enables lossless packet delivery over routed networks, which is especially important for technologies like RoCEv2 (RDMA over Converged Ethernet v2), where even minimal packet loss can cause significant performance degradation.

DSCP-Based PFC Process over a Layer 3 Routed Interface (Example Scenario)

This example illustrates how DSCP-based Priority Flow Control (PFC) operates across a routed Layer 3 fabric during

congestion. We walk through a four-step process, beginning with buffer overflow and ending with traffic pausing on the correct priority queue.

Step 1: Buffer Overflow on Rail Switch C (Egress to GPU-3, Host 3)

In a GPU cluster, multiple GPUs are sending high-throughput RDMA traffic to GPU on Host-3. In Figure 11-4 Rail Switch C is responsible for forwarding traffic toward GPU-3. The egress interface on Switch C (E12/24) that connects to GPU-3 becomes congested. Due to the overflow of egress

queue 3, packets from ingress queue 3 on interface E3/24 cannot be placed into egress queue 3.

Step 2: xOFF Threshold Exceeded

Priority queue 3 of has two configured thresholds:

- xOFF threshold: Triggers a pause when buffer usage exceeds this level.
- xON threshold: Triggers a resume when the buffer has drained sufficiently.

Once priority queue 3 on ingress interface E3/24 exceeds its xOFF threshold, the switch takes immediate action to prevent packet loss by generating a PFC pause message targeted at the sender. The sender in this case is Spine Switch 1, which is

sending traffic to Rail Switch C, over interface E3/24, for delivery to GPU-3.

Step 3: Generating a PFC Pause Frame (MAC Control Frame)

To pause the sender, Rail Switch C generates an Ethernet MAC Control frame with:

- Ethertype 0x8808: This indicates a MAC Control frame, used for pause-related operations (standardized in IEEE 802.3x). Inside this frame, a PFC opcode (0x0101) specifies it's a Priority-based Pause (PFC) message.
- Class Enable Vector (CEV): This 8-bit field indicates which priority queues should be paused. Each bit corresponds to one of the 8 possible traffic classes (0–7). For example, if bit 3 is set to 1, it tells the sender to pause traffic for priority queue 3 only, while all other bits remain 0. In our case, the CEV is 0000 1000. Note that the rightmost bit represents queue 0.
- Quanta Field(s): For each enabled priority (bit set to 1), a corresponding quanta value is specified. This value defines the duration of the pause, measured in units of 512 bit times.

For a 400 Gbps interface:

- 1 bit time = 1 / 400,000,000,000 seconds ≈ 2.5 picoseconds

- 1 quanta = 512 × 2.5 ps = 1.28 nanoseconds
- If the pause quanta is set to maximum value 0xFFFF (65535), the pause duration is roughly 83.9 microseconds.

This pause frame is sent back to the sender Spine Switch 1. Since the DSCP-based classification maps back to priority queue 3, and the switches share the same mapping, Spine Switch 1 will interpret this correctly.

Step 4: Spine Switch 1 Pauses Transmission on Priority Queue 3

Upon receiving the PFC frame on its ingress interface E3/24 connected to Rail Switch C, Spine Switch 1 examines the class enable vector.

- Since bit 3 is set, the switch knows to pause transmission of all frames mapped to priority queue 3 (DSCP value 24 in our example) on egress interface E3/24.

Traffic for other priority queues continues unaffected.

Spine Switch 1 holds off transmission of priority 3 traffic until it receives a subsequent PFC frame with quanta = 0, indicating “resume,” or a pause duration timeout occurs, after which the switch resumes sending unless another pause is received.

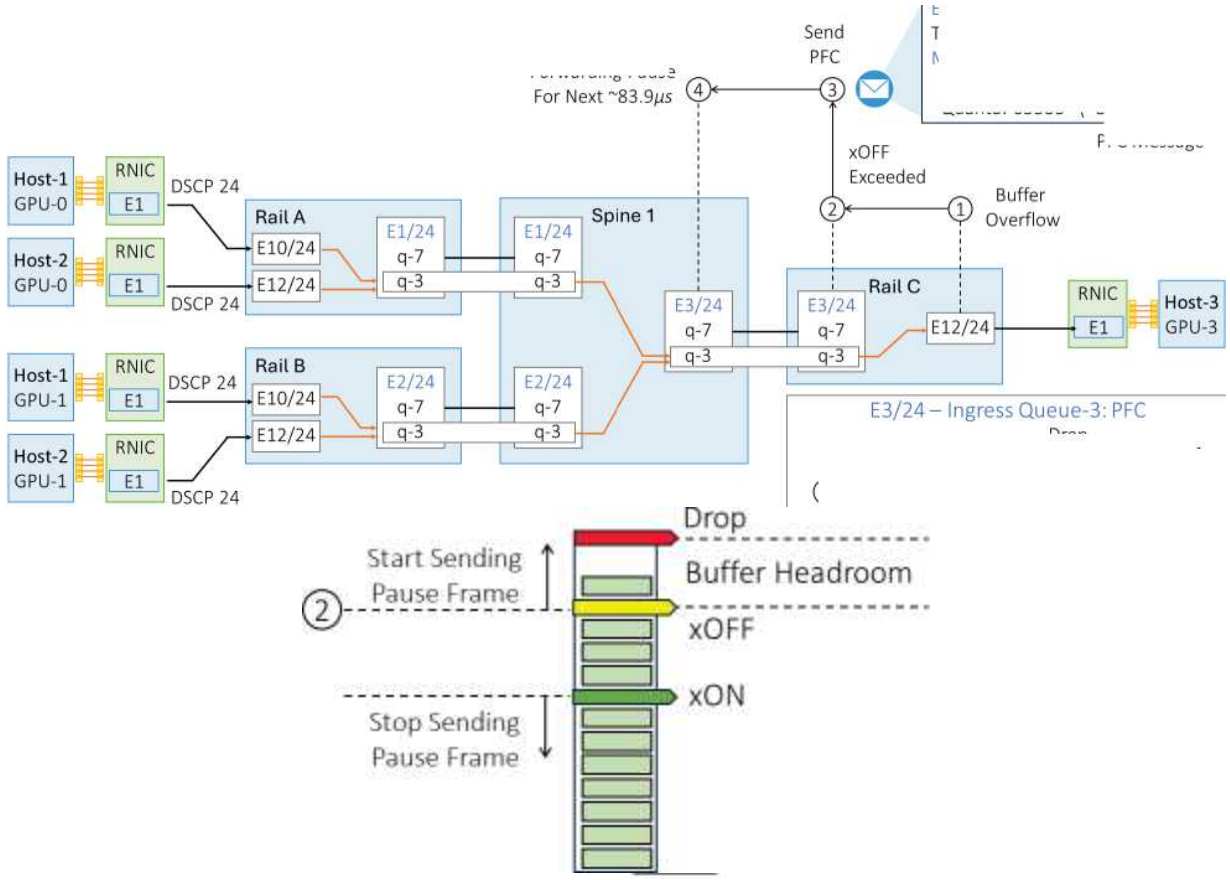


Figure 11-4: *Priority Flow Control – Pause Frame.*

The following example shows how Priority Flow Control (PFC) events can cascade upstream when congestion persists in a routed Layer 3 fabric. This scenario builds on the earlier case, where Spine Switch 1 paused traffic to Rail Switch C. Now, we observe how that pause affects traffic originating from Rail Switches A and B.

Step 5: Congestion on Spine Switch 1 Egress Queue to Rail Switch C

As described in the previous figure, Spine Switch 1 received a PFC frame from Rail Switch C and responded by pausing traffic

on priority queue 3 on its egress interface E3/24 (towards Rail Switch C). Because this interface is no longer sending traffic, frames destined for GPU-3 via Rail Switch C begin to accumulate in Spine Switch 1's egress queue 3. This build-up causes backpressure that impacts the ingress side of the switch.

Step 6: xOFF Threshold Exceeded on Spine Switch 1 Ingress Interfaces

Spine Switch 1 receives incoming traffic from Rail Switch A (interface E1/24) and Rail Switch B (interface E2/24). Both switches are sending traffic mapped to priority queue 3 (e.g., DSCP 24). As the egress queue to Rail Switch C becomes full and cannot drain, the corresponding ingress buffers on interfaces E1/24 and E2/24 also begin to fill up, specifically for queue 3. Eventually, the xOFF thresholds on both ingress interfaces are exceeded, indicating that congestion is now impacting the reception of new packets on these ports.

Step 7: Spine Switch 1 Sends PFC Pause Frames to Rail Switch A and B

To avoid dropping packets due to ingress buffer overflow, Spine Switch 1 generates PFC MAC Control frames on both E1/24 and E2/24. The class enable vector has bit 3 set, instructing the sender to pause traffic corresponding to priority queue 3. A suitable quanta value is included to define the pause duration.

These control frames travel back to Rail Switch A and Rail Switch B respectively.

Step 8: Rail Switches A and B Pause Queue 3 Traffic to Spine Switch 1

Upon receiving the PFC frames, both Rail Switch A and Rail Switch B interpret the class enable vector and pause all traffic mapped to priority queue 3 (e.g., DSCP 24), still forwarding traffic on other priority queues unaffected. This marks the upstream propagation of congestion: a single bottleneck on the path to GPU-3 can trigger PFC reactions all the way back to multiple source switches.

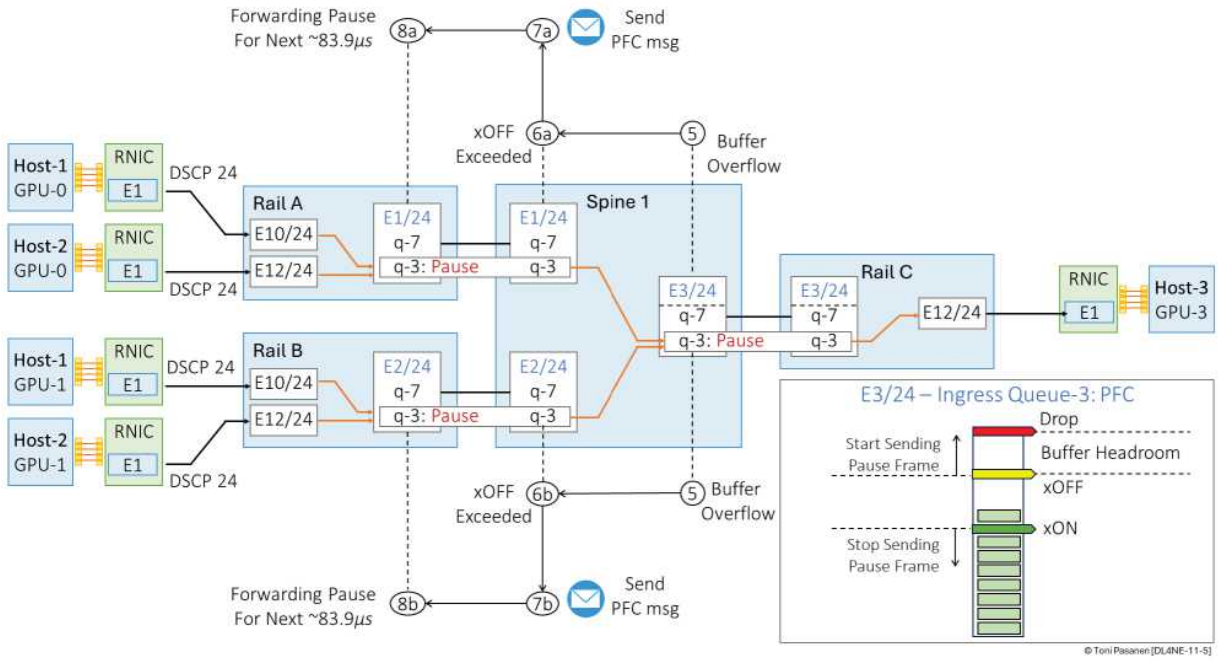


Figure 11-5: *Priority Flow Control – Cascading Effect.*

Steps 9a – 14: Downstream Resume and Congestion Recovery

Figure 11-6 illustrates how the PFC-based congestion recovery process extends from Rail Switches A and B all the way to the GPU NICs, while simultaneously resolving the initial congestion at Rail Switch C.

As a result of the earlier PFC pause frames:

- Rail Switch A and Rail Switch B have paused sending priority queue 3 traffic to Spine Switch 1.
- In turn, Spine Switch 1 has paused its own egress traffic toward Rail Switch C on interface E3/24.

This pause allows queue 3 on Rail Switch C's egress interface E12/24 (toward GPU-3) to drain, as no new traffic is arriving, and the GPU continues to consume incoming data.

Once the buffer utilization for priority queue 3 drops below the configured xON threshold, Rail Switch C initiates congestion recovery.

- It sends a MAC Control Frame (Ethertype 0x8808) back to Spine Switch 1.
- The class enable vector has bit 3 set (indicating priority queue 3).
- The quanta value is set to 0, signaling that it is now safe to resume transmission.

Upon receiving this resume message, Spine Switch 1 can begin sending traffic again on priority queue 3, restoring throughput toward GPU-3 and continuing the flow of RDMA traffic through the network. This recovery mechanism operates consistently across the entire AI fabric.

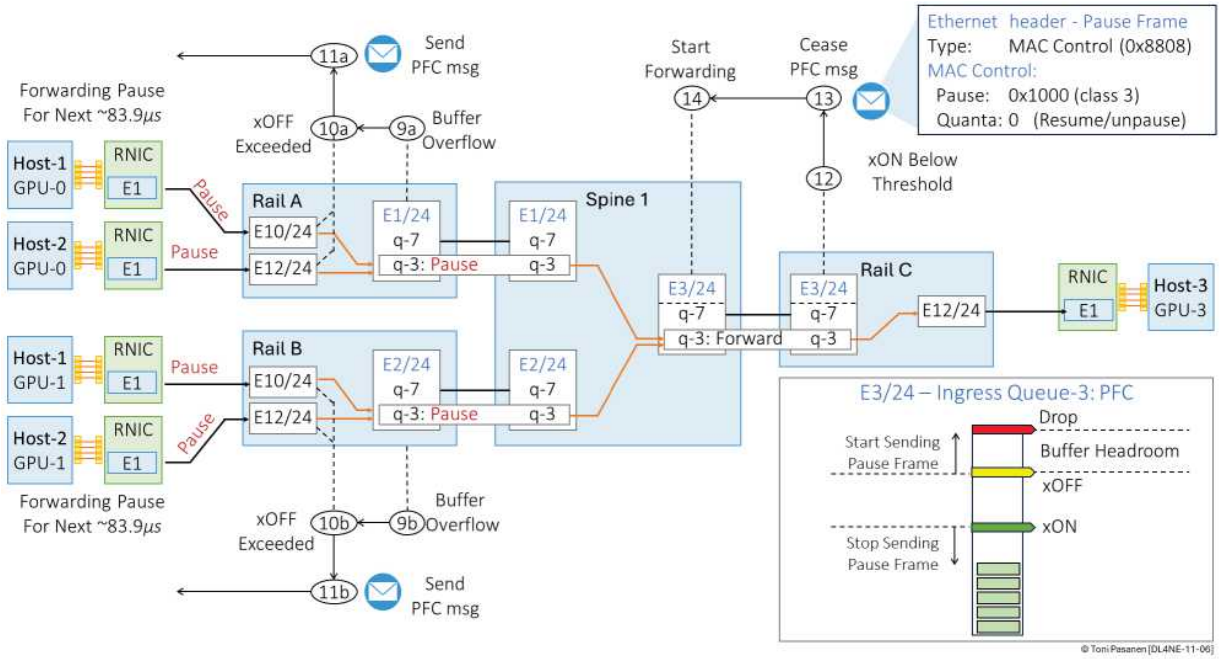


Figure 11-6: *Priority Flow Control – PCIe Bus Congested: Cascading Effect.*

LLDP WITH DCBX

PFC negotiation is performed using the Link Layer Discovery Protocol (LLDP), which carries Data Center Bridging eXchange (DCBX) TypeLength-Value (TLV) structures. At the time of writing, DCBX exists in two versions: IEEE and CEE. The IEEE mode (defined in 802.1Qbb and 802.1Qaz) is standards-based and supported by most modern data center switches from various vendors. This mode is also known as DCBXv2. Some older Cisco Nexus models support only the Cisco/Converged Enhanced Ethernet (CEE) mode. Capture 11-1 shows the packet format of a standards-based IEEE DCBX TLV within an LLDP message.

Ethernet II:

Source MAC:

Destination MAC:

Link Layer Discovery Protocol

Chassis Subtype = MAC address, ID

Port Subtype: = Interface name, ID

Time To Live: 120 sec

IEEE 801.1 - ETS Configuration

IEEE 801.1 - ETS Recommendation

IEEE 801.1 - Priority Flow Control Configuration

<snipped for brevity>

1000 = Max	PFC	Enabled Traffic Classes	
...0 = PFC	for	Priority 0 -	Disabled
..0. = PFC	for	Priority 1 -	Disabled
.0.. = PFC	for	Priority 2 -	Disabled

....1... = PFC for Priority 3 - Enabled

<snipped for brevity>

IEEE 802.1 - Application Protocol

End of LLDPDU

Capture 11-1: *PCF: LLDP with IEEE DBCXv2 TLV* .

DATA CENTER QUANTIZED CONGESTION NOTIFICATION (DCQCN)

Data Center Quantized Congestion Notification (DCQCN) is a hybrid congestion control method. DCQCN brings together both Priority Flow Control (PFC) and Explicit Congestion Notification (ECN) so that we can get high throughput, low latency, and lossless delivery across our AI fabric. In this approach, each mechanism plays a specific role in addressing different aspects of congestion, and together they create a robust flowcontrol system for RDMA traffic.

DCQCN tackles two main issues in large-scale RDMA networks:

1. Head-of-Line Blocking and Congestion Spreading: This is caused by PFC's pause frames, which stop traffic across switches.
2. Throughput Reduction with ECN Alone: When the ECN feedback is too slow, packet loss may occur despite the rate adjustments.

DCQCN uses a two-tiered approach. It applies ECN early on to gently reduce the sending rate at the GPU NICs, and it uses PFC as a backup to quickly stop traffic on upstream switches (hop-by-hop) when congestion becomes severe.

How DCQCN Combines ECN and PFC

DCQCN carefully combines Explicit Congestion Notification (ECN) and Priority Flow Control (PFC) in the right sequence:

Early Action with ECN:

When congestion begins to build up, the switch uses WRED thresholds (minimum and maximum) to mark packets. This signals the sender to gradually reduce its transmission rate. As a result, the GPU NIC slows down, and traffic continues flowing—just at a reduced pace—without abrupt pauses.

Backup Action with PFC:

If congestion worsens and the queue continues to grow, the buffer may reach the xOFF threshold. At this point, the switch sends PFC pause frames hop by hop to upstream devices. These devices respond by temporarily stopping traffic for that specific priority queue, helping prevent packet loss.

Resuming Traffic:

Once the buffer has drained and the queue drops below the xON threshold, the switch sends a resume message (a PFC frame with a quanta value of 0). This tells the upstream device it can start sending traffic again.

Why ECN Must Precede xOFF

It is very important that the ECN thresholds (WRED minimum and maximum) are used before the xOFF threshold is reached

for three main reasons:

Graceful Rate Adaptation: Early ECN marking helps the GPU NIC (sender) reduce its transmission rate gradually. This smooth adjustment avoids sudden stops and leads to more stable traffic flows.

Avoiding Unnecessary PFC Events: If the sender adjusts its rate early with ECN feedback, the buffers are less likely to fill up to the xOFF level. This avoids the need for abrupt PFC pause frames that can cause head-of-line blocking and backpressure on the network.

Maintaining Fabric Coordination: With early ECN marking, the sender receives feedback before congestion becomes severe. While the ECN signal is not shared directly with other switches, the sender's rate adjustment helps reduce overall pressure on the network fabric.

What Happens If xOFF Is Reached Before ECN Marking?

Imagine that the ingress queue on Spine Switch 1 (from Rail Switch A) fills rapidly without ECN marking:

Sudden Pause: The buffer may quickly hit the xOFF threshold and trigger an immediate PFC pause.

Downstream Effects: An abrupt stop in traffic from Rail Switch A leads to sudden backpressure. This can cause head-of-line blocking and disturb GPU communication, leading to performance jitter or instability at the application level.

Oscillations: When the queue finally drains and reaches the xON threshold, traffic resumes suddenly. This can cause recurring congestion and stop-and-go patterns that hurt overall performance.

By allowing ECN to mark packets early, the network gives the sender time to reduce its rate smoothly. This prevents abrupt stops and helps maintain a stable, efficient fabric.

Figure 11 recaps how the example DCQCN process works:

Time t1: (1) Traffic associated with priority queue 3 on Rail-1's egress interface 1 crosses the WRED minimum threshold.

Time t2: (2) Rail-1 begins randomly marking ECN bits as 11 on packets destined for GPU-0 on the Host-3.

Time t3: (3) The RDMA NIC starts sending CNP messages to the sender GPU-1 on Host-1.

Time t4: (4) In response to the CNP message, the sending GPU-0 on Host-1 reduces its transmission rate by holding packets longer in its egress queue. (5) At the same time, egress queue 3 on Rail-1 remains congested. (6) Since packets cannot be forwarded from ingress interface 2 to egress interface 1's queue 3, ingress interface 3 also becomes congested, eventually crossing the PFC xOFF threshold.

Time t5: (7) As a result, Rail-1 sends a PFC xOFF message to Spine-A over Inter-Switch Link 3. (8) In response, Spine-A halts forwarding traffic for the specified pause duration.

Time t6: (9) Due to the forwarding pause, the egress queue of interface 3 on Spine-A becomes congested, which in turn (10) causes congestion on its ingress interface 2.

Time t7: (11) The number of packets waiting in egress queue 3 on interface 1 of Rail-1 drops below the WRED minimum threshold. (12) This allows packets from the buffer of interface 3 to be forwarded.

Time t8: (13) The packet count on ingress interface 3 of Rail-1 falls below the PFC xON threshold, triggering the PFC resume/unpause message to Spine-A. (14) Spine-A resumes forwarding traffic to Rail-1.

After the PFC resume message is sent, Spine-A starts forwarding traffic again toward Rail-1. The congestion on Spine-A's interface 3 gets cleared as packets leave the buffer. This also helps the ingress interface 2 on Spine-A to drain. On Rail-1, as interface 1 can now forward packets, queue 3 gets more room, and the flow to GPU-0 becomes smoother again.

The RDMA NIC on the sender GPU monitors the situation. Since there are no more CNP messages coming in, the GPU slowly increases its sending

rate. At the same time, the ECN marking on Rail-1 stops, as queue lengths stay below the WRED threshold. Traffic flow returns to normal, and no more PFC pause messages are needed.

The whole system stabilizes, and data can move again without delay or packet loss.

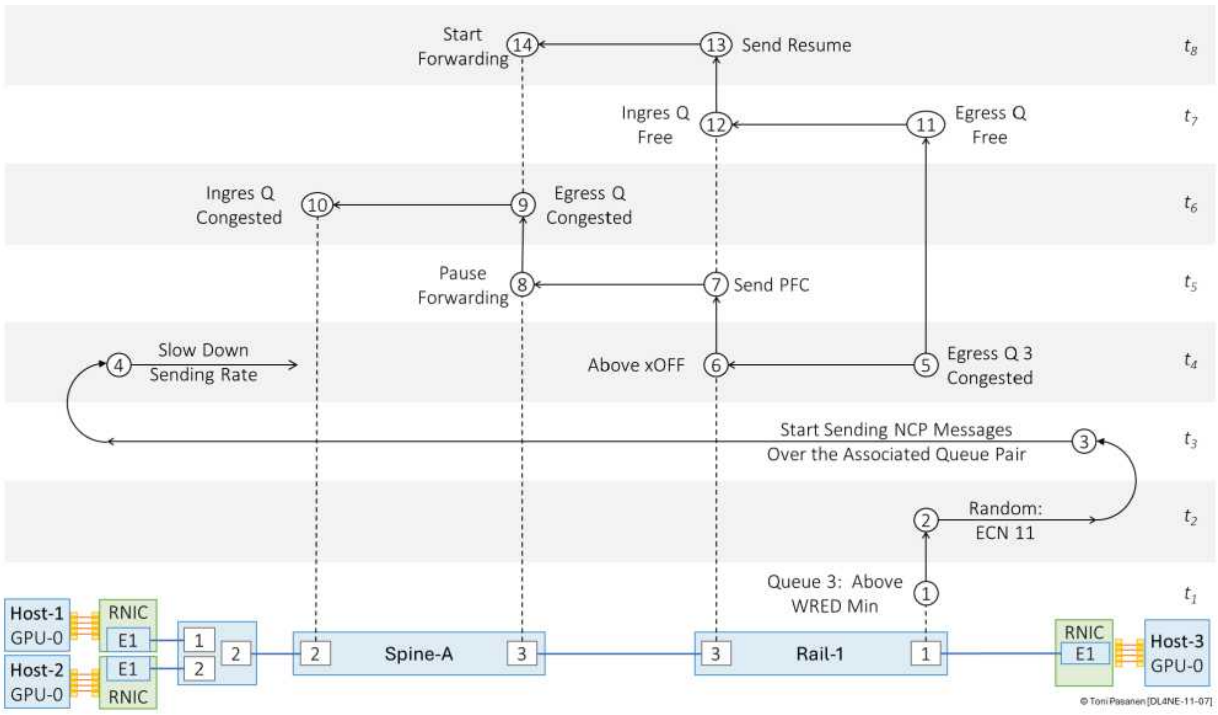


Figure 11-7: *DCQCN: ECN and PFC Interaction .*

DCQCN Configuration

Figure 11-8 shows the six steps to enable DCQCN on a switch. The figure assumes that the RDMA NIC marks RoCEv2 traffic with DSCP 24.

First, we classify the packets based on the DSCP value in the IPv4 header. Packets marked with DSCP 24 are identified as RoCEv2 packets, while packets marked with DSCP 48 are classified as CNP.

After classification, we add an internal QoS label to the packets to place them in the correct output queue. The mapping between internal QoS labels and queues is fixed and does not require configuration.

Next, we define the queue type, allocate bandwidth, and set ECN thresholds. After scheduling is configured, we enable PFC and set its threshold values. A common rule of thumb for the relationship between ECN and PFC thresholds is: xON < WRED Min < WRED Max < xOFF.

To apply these settings, we enable them at the system level. Finally, we apply the packet classification to the ingress interface and enable the PFC watchdog on the egress interface. Because PFC is a sub-TLV in the LLDP Data Unit (LLDPDU), both LLDP and PFC must be enabled on every interswitch link.

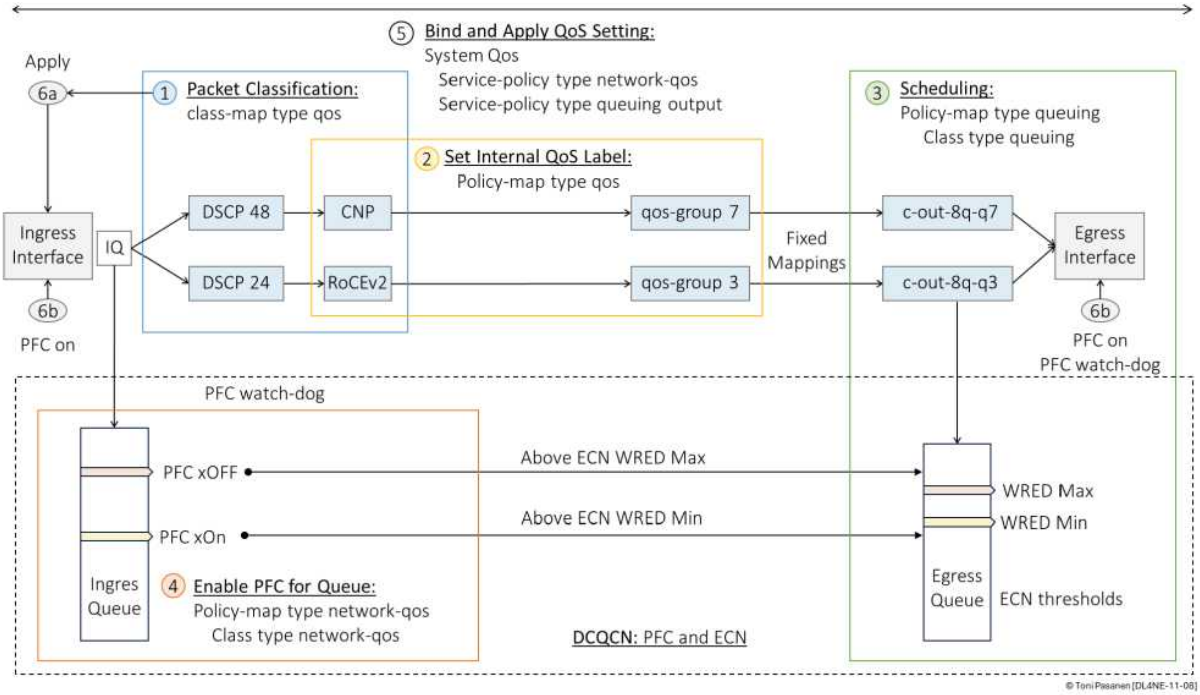


Figure 11-8: *Applying DCQCN to Switch.*

Step 1: Packet Classification

The classification configuration is used to identify different types of traffic based on their DSCP values. In our example we

have one for RoCEv2 traffic and another for Congestion Notification Packets (CNP). The “class-map type qos match-any ROCEv2” line defines a class map named “ROCEv2” that matches any packet marked with DSCP value 24, which is commonly used for RDMA traffic. Similarly, the “class-map type qos match-any CNP” defines another class map named “CNP” that matches packets marked with DSCP value 48, typically used for congestion signaling in RDMA environments. These class maps serve as the foundation for downstream policies, enabling differentiated handling of traffic types. Note that the names “ROCEv2” and “CNP” are not system-reserved; they are simply user-defined labels that can be renamed, as long, as the references are consistent throughout the configuration.

```
class-map type qos match-any ROCEv2
match dscp 24
class-map type qos match-any CNP
match dscp 48
```

Example 11-1: *Classification*.

Step 2: Internal QoS Label for Queueing

The marking configuration assigns internal QoS labels to packets that have already been classified. This is done using a policy map named QOS_CLASSIFICATION, which refers to the previously defined class maps. Within this policy, packets that match the “ROCEv2” class are marked with *qos-group* 3, and

those matching the “CNP” class are marked with *qos-group* 7. Any other traffic that doesn't fit these two categories falls into the default class and is marked with *qos-group* 0. These QoS groups are internal identifiers that the switch uses in later stages for queuing and scheduling, to decide how each packet should be treated. Just like class maps, the name of the policy map itself is user-defined and can be anything descriptive, provided it is correctly referenced in other parts of the configuration.

policy-map type **qos** QOS_CLASSIFICATION

class ROCEv2

set qos-group 3 class CNP

set qos-group 7 class class-default

set qos-group 0

Example 11-2: *Marking*.

Step 3: Scheduling

The queuing configuration defines how traffic is scheduled and prioritized on the output interfaces, based on the internal QoS groups that were assigned earlier. This is handled by a policy map named “QOS_EGRESS_PORT,” which maps traffic to different hardware output queues. Each queue is identified by a class, such as c-out-8q-q7 (fixed names: 8q = eight queues, q7 = queue number 7). For example, queue 7 is configured with priority level 1, which gives it strict priority over all other traffic. Queue 3 is assigned *bandwidth remaining percent* 50, meaning

that it is guaranteed half of the remaining bandwidth after strict-priority traffic has been serviced. In addition to bandwidth allocation, queue 3 includes congestion management features through the *random-detect* command. This enables Weighted Random Early Detection (WRED), a mechanism that helps avoid congestion by randomly mark packets as queue depth increases. The *minimum-threshold* and *maximum-threshold* define the WRED minimum and maximum values (from 150 KB to 3000 KB) at which packets begin marked. The *drop-probability* 7 determines the likelihood of packet mark when the maximum threshold is reached, with higher numbers indicating higher marking rates. The *weight* 0 setting controls how queue size is averaged. A weight of 0 means use instantaneous queue depth (no averaging). Finally, *ecn* enables Explicit Congestion Notification, allowing network devices to signal congestion without dropping packets, without the *ecn* option switch drops packet based on WRED min/max values. The remaining queues are configured with either zero percent of remaining bandwidth, effectively disabling

them for general use, or with a share of the remaining bandwidth. This queuing policy ensures that RoCEv2 traffic receives adequate resources with congestion feedback, while CNP messages always get through with strict priority.

policy-map type queuing QOS_EGRESS_PORT

class type queuing c-out-8q-q6

bandwidth remaining percent 0

...

class type queuing c-out-8q-q3

bandwidth remaining percent 50

random-detect minimum-threshold 150 kbytes maximum-threshold 3000 kbytes drop-probability 7 weight 0 ecn

...

class type queuing c-out-8q-q7

priority level 1

Example 11-3: *Queuing (Output Scheduling).*

Step 4: Enable PFC for Queue

The Network QoS configuration defines the low-level, hardware-based characteristics of traffic handling within the switch, such as enabling lossless behavior and setting the maximum transmission unit (MTU) size for each traffic class. In this example, the *policy-map type network-qos qos_network* is used to configure how traffic is handled inside the switch fabric. Under this policy, the *class type network-qos c-8q-nq3* is associated with *pause pfc-cos 3*, which enables Priority Flow Control (PFC) on Class of Service (CoS) 3. This is critical for RoCEv2 traffic, which depends on a lossless transport layer. The MTU is also defined here, with bytes (jumbo frame) set for class 3 traffic.

policy-map type network-qos qos_network class type network-qos c-8q-nq3

mtu 9216

pause pfc-cos 3

Example 11-4: *Queuing (Output Scheduling).*

Priority Flow Control Watchdog

The Priority Flow Control (PFC) watchdog is a mechanism that protects the network from traffic deadlocks caused by stuck PFC pause frames. In RDMA environments like RoCEv2, PFC is used to create lossless classes of traffic by pausing traffic flows instead of dropping packets. However, if a device fails to release the pause or a misconfiguration causes PFC frames to persist, traffic in the affected class can become permanently blocked, leading to what is called a "head-of-line blocking" condition. To mitigate this risk, the priority-flow-control watch-dog-interval on command enables the PFC watchdog feature. When enabled, the switch monitors traffic in each PFC-enabled queue for signs of persistent pause conditions. If it detects that traffic has been paused for too long, indicating a potential deadlock, it can take corrective actions, such as generating logs, resetting internal counters, or even discarding paused traffic to restore flow.

priority-flow-control watch-dog-interval on **Example 11-5:** *Priority Flow Control (PFC) Watchdog.*

Step 5: Bind and Apply QoS Settings

System-level QoS policies bind all the previously defined QoS components together and activate them across the switch. This is done using the *system qos* configuration block, which applies the appropriate policy maps globally. The *service-policy type network-qos qos_network* command activates the network-qos policy defined earlier, ensuring that MTU sizes and PFC configurations are enforced across the switch fabric. The command *service-policy type queuing output* QOS_EGRESS_PORT applies the queuing policy at the output interface level, enabling priority queuing, bandwidth allocation, and congestion management as traffic exits the switch. These system-level bindings are essential because, without them, the individual QoS policies, classification, marking, queuing, and fabric-level configuration, would remain inactive. By applying the policies under *system qos*, the switch is instructed to treat

traffic according to the rules and priorities defined in each policy map. This final step ensures end-to-end consistency in QoS behavior, from ingress classification to fabric transport and egress scheduling, providing a complete and operational quality-of-service framework tailored for latency-sensitive, lossless applications like RoCEv2.

system qos

service-policy type network-qos qos_network

service-policy type queuing output QOS_EGRESS_PORT

Example 11-6: *Priority Flow Control (PFC) Watchdog*.

Step 6: Interface-Level Configuration

The interface-level configuration attaches the previously defined QoS policies and enables PFC-specific features for a given port. In our example, the configuration is applied to Ethernet2/24, but the same approach can be used for any interface where you need to enforce QoS and PFC settings. The first command, *priority-flow-control mode auto*, enables Priority Flow Control (PFC) on the interface in auto-negotiation mode. This means the interface will automatically negotiate PFC with its link partner, allowing for lossless traffic handling by pausing specific traffic classes instead of dropping packets. The *priority-flow-control watch-dog* command enables the PFC watchdog for this interface, which ensures that if any PFC pause frames are stuck or persist for too long, the watchdog will take corrective action to prevent a deadlock situation. This helps maintain the overall health of the network by preventing traffic congestion or blockages due to PFC-related issues.

Lastly, the *service-policy type qos input* QOS_CLASSIFICATION command applies the QoS classification policy on incoming traffic, ensuring that packets are classified and marked according to their DSCP values as defined in the QOS_CLASSIFICATION policy. This classification enables downstream QoS treatment, including proper queuing, scheduling, and priority handling.

interface Ethernet 2/24

priority-flow-control mode auto

priority-flow-control watch-dog

```
service-policy type qos input QOS_CLASSIFICATION
```

Example 11-7: *Interface Level Configuration*.

REFERENCES

[1] Nemanja Kamenica, Cisco, Network Best Practices for Artificial Intelligence Data Centre, Cisco Live, BRKDCN-2921,

https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2024/pdf/BR KDCN-2921.pdf

[2] Class of Service User Guide (Routers and EX9200 Switches) Understanding PFC Using DSCP at Layer 3 for Untagged Traffic, January 2021.

https://www.juniper.net/documentation/us/en/software/junos/cos/topics/con cept/cos-lossless-l3-dscp-pfc-understanding.html

[3] Class of Service User Guide (Routers and EX9200 Switches) Configuring DSCP-based PFC for Layer 3 Untagged Traffic, February 18, 2021.

https://www.juniper.net/documentation/us/en/software/junos/cos/topics/tas k/cos-configuring-l3-dscp-pfc.html

[4] Ethernet flow control, Wikipedia, https://en.wikipedia.org/wiki/Ethernet_flow_control.

[5] Explicit Congestion Notification, Wikipedia, https://en.wikipedia.org/wiki/Explicit_Congestion_Notification

[6] K. Ramakrishnan - TeraOptic Networks, et al., “The Addition of Explicit Congestion Notification (ECN) to IP”, September 2001

[7] Michael Witte, Understanding Data Center Quantized Congestion Notification (DCQCN), World Wide Technology, Article June 17, 2024. https://www.wwt.com/article/understanding-data-center-quantized-congestion-notification-dcqcn

CHAPTER 12: FLOW, FLOWLET, AND PACKET-BASED LOAD BALANCING

INTRODUCTION

Though BGP supports the traditional Flow-based Layer 3 Equal Cost Multi-Pathing (ECMP) traffic load balancing method, it is not the best fit for a RoCEv2-based AI backend network. This is because GPU-to-GPU communication creates massive elephant flows, which RDMA-capable NICs transmit at line rate. These flows can easily cause congestion in the backend network.

In ECMP, all packets of a single flow follow the same path. If that path becomes congested, ECMP does not adapt or reroute traffic. This leads to uneven bandwidth usage across the network. Some links become overloaded, while others remain idle. In AI workloads, where multiple high-bandwidth flows occur at the same time, this imbalance can degrade performance.

Deep learning models rely heavily on collective operations like all-reduce, all-gather, and broadcast. These generate dense traffic patterns between GPUs, often at terabit-per-second speeds. If these flows are not evenly distributed, a single congested path can slow down the entire training job.

This chapter introduces two alternative load balancing methods to traditional Flow-Based with Layer 3 ECMP: 1) Flowlet-Based Load Balancing with Adaptive Routing, and 2) Packet-Based Load Balancing with Packet Spraying. Both aim to

improve traffic distribution in RoCEv2-based AI backend networks, where conventional flow-based routing often leads to congestion and underutilized links. These advanced methods are designed to handle the unique traffic patterns of AI workloads more efficiently.

RDMA WRITE OPERATION

Before we explore the load balancing solution, let's first walk through a simplified example of how the RDMA WRITE memory copy operation works. In Figure 12-1, we have two GPU servers: Host 1 and Host 2, each with one GPU. By this point, the memory has already been allocated and registered, and the Queue Pair (QP) has been created on both sides, so the data transfer can begin.

On GPU-0 of Host 1, gradients are stored in memory regions highlighted in green, orange, and blue. Each colored section represents a portion of local memory that will be written to GPU-0 on Host 2. To transfer the data, the RDMA NIC on Host 1 splits the write operation into three flowlets (green, orange, and blue). Rather than sending the entire data block as a single continuous stream, each flowlet is treated as a segment of the same RDMA Write operation.

RDMA Write First

The first message carries the RDMA Extended Transport Header (RETH) in its payload. This header tells the receiving RDMA NIC where in the remote memory the incoming data should be written. In our example, data from memory block 1B of GPU-0 on Host 1 is written to memory block 2C of GPU-0 on Host 2.

The RETH contains the R_Key, which gives permission to write to the remote memory region. It also includes the length of the data being transferred and the virtual address of the target memory location on Host 2.

The operation code in the InfiniBand Base Transport Header (IBTH) is set to RDMA Write First, indicating that this is the first message in the sequence. The IBTH also describes the Partition Key (interface identifier), the Destination Queue Pair number, and the Packet Sequence Number (PSN) that helps ensure packets are processed in the correct order.

When this first packet arrives at Host 2, the RDMA NIC uses the Virtual Address information in the RETH header to write the payload directly into memory block 2C.

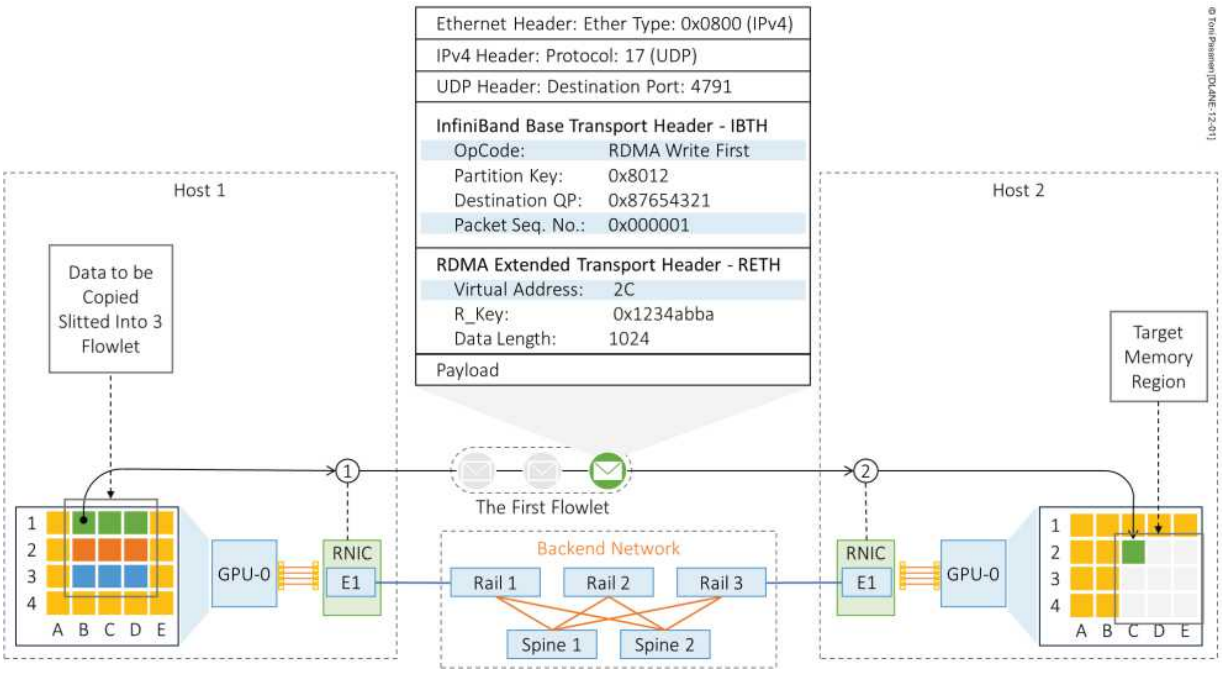


Figure 12-1: *RDMA WRITE First.*

RDMA Write Middle

The second message has the opcode RDMA Write Middle and PSN 2, which tells the receiver that this packet comes after the first one with PSN 1. The payload of this Flowlet is written right after the previous block, into memory block 2D on Host 2. The RDMA NIC ensures that the order is maintained based on PSNs, and it knows exactly where to place the data thanks to the original offset from the first packet.

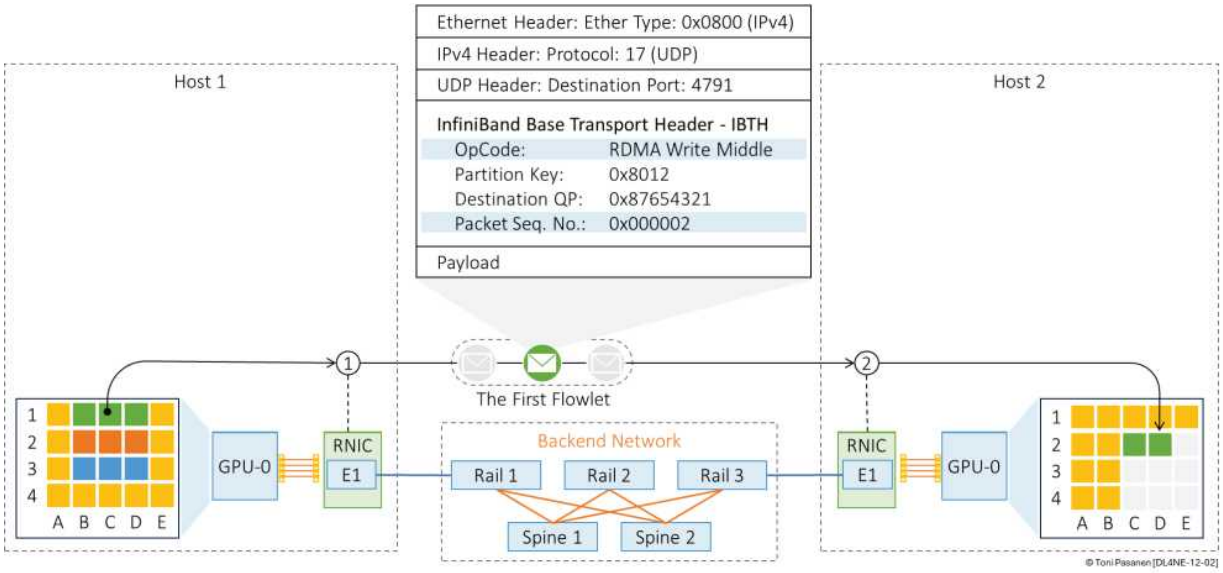


Figure 12-2: *RDMA WRITE Middle*.

RDMA Write Last

The third message has the opcode RDMA Write Last, indicating that this is the final message in the sequence. With PSN 3, it follows directly after PSN 2. The payload in this packet is written into memory block 2E, which comes directly after 2D.

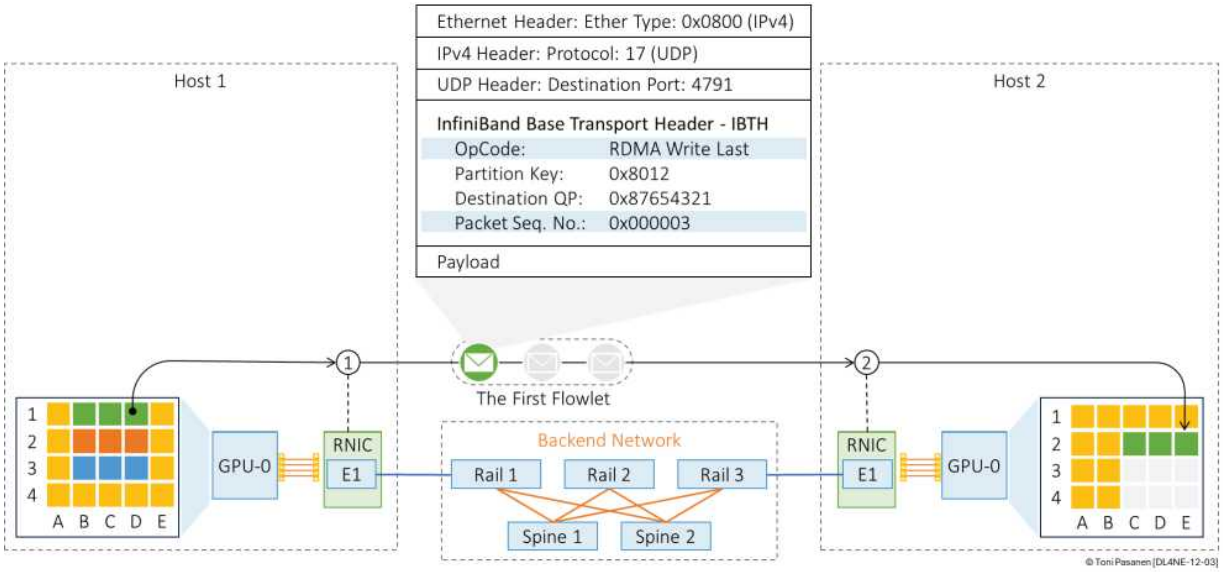


Figure 12-3: *RDMA WRITE Last.*

In a multi-packet RDMA Write operation, each Flowlet represents a continuous block of data being transferred from the source GPU to the destination GPU. Data within packets must arrive in the correct order because only the first packet includes the full addressing information in the RDMA Extended Transport Header (RETH). This header tells the receiver where in memory the data should be written.

Packets marked as RDMA Write Middle and RDMA Write Last depend on this information and must follow the sequence defined by the Packet Sequence Numbers (PSNs). If packets are delivered out of order, the receiving RDMA NIC cannot process them immediately. Instead, it must hold them in memory and wait for the missing earlier packets to arrive. This buffering increases memory usage and processing overhead. In high-

speed environments, this can lead to performance degradation or even packet drops, especially when buffers fill up under heavy

load.

FLOW-BASED LOAD BALANCING WITH LAYER 3 ECMP

Figure 12-4 depicts the problem with flow-based load balancing when used in an AI fabric backend network. In our example, we have four hosts, each equipped with two GPUs: GPU-1 and GPU-2. The RDMA NICs connected to GPU-1s are linked to switch Rail-1, and the RDMA NICs connected to GPU-2s are linked to Rail-2. Traffic between NICs on Rail-1 and Rail-2 is forwarded through either Spine-1 or Spine-2.

We use a basic data parallelization strategy, where the training dataset is divided into mini-batches and distributed across all eight GPUs. To keep the example simple, Figure 12-4 only shows the all-reduce gradient synchronization flow from the GPU-1s on Hosts 1, 2, and 3 to the GPU-2 on Host 4. In real-world training, a full-mesh all-reduce operation takes place between all GPUs.

As a starting point, the GPU-1s on the three leftmost hosts begin the RDMA process to copy data from their memory to the memory of GPU-2 on Host 4. These GPU-1s are all connected to Rail-1. Instead of sending one large flow, the RDMA NICs split the data into flowlets, small bursts of data from the larger transfer. These flowlets arrive at the Rail-1 switch, where the 5tuple L3 ECMP hash algorithm unfortunately selects the same uplink for all three flows.

Since the switch cannot forward all the data at wire speed, it stores some of the packets in the buffer, causing congestion. Similar congestion may also occur at the spine switches. As explained earlier in Chapter 12, egress buffer overflow may trigger ECN (Explicit Congestion Notification) and PFC (Priority Flow Control) mechanisms to prevent packet loss.

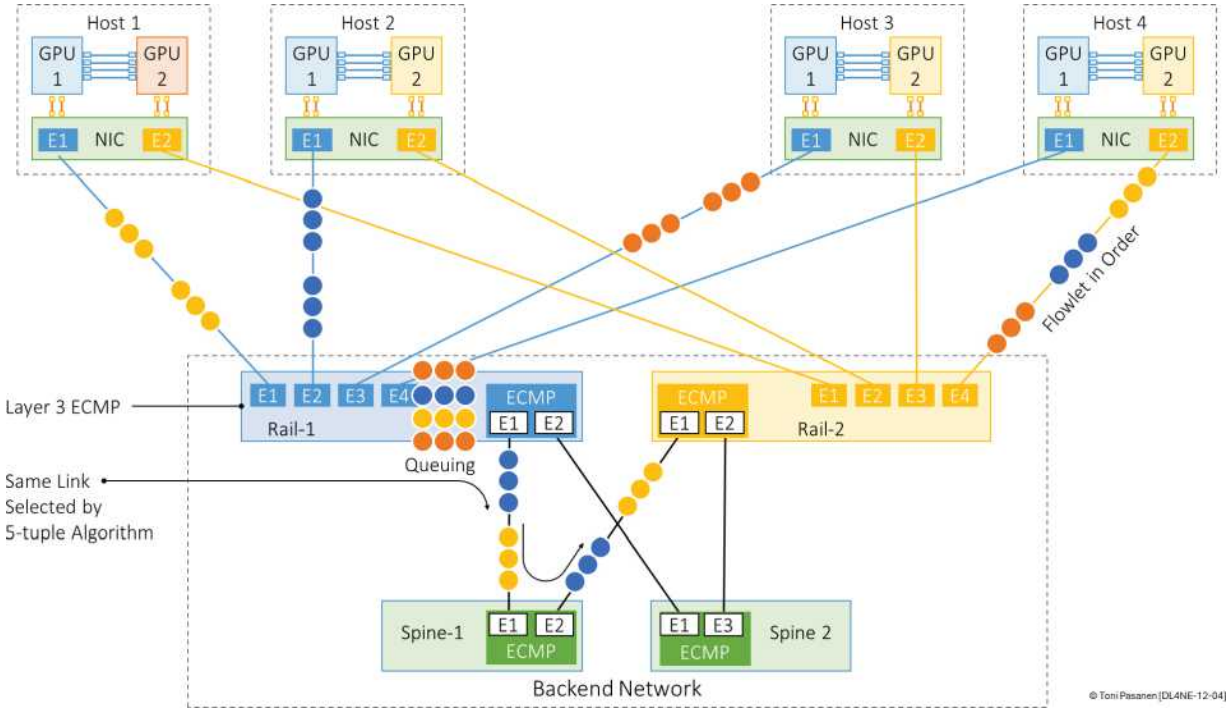


Figure 12-4: *Layer 3 Load balancing.*

FLOWLET-BASED LOAD BALANCING WITH ADAPTIVE ROUTING

Adaptive routing is a dynamic method that actively monitors link utilization and reacts to network congestion in real time. In Figure 12-5, the 5-tuple hash algorithm initially selects the same uplink for all flowlets, just like in the previous example. However, once the utilization of the interswitch link between Rail-1 and Spine-1 goes over threshold, the adaptive routing mechanism detects the increased load and starts redirecting some of the flowlets to an alternate, less congested path, through Spine-2.

By distributing the flowlets across multiple paths, adaptive routing helps to reduce buffer buildup and avoid potential packet drops. This not only improves link utilization across the fabric but also helps maintain consistent throughput for time-sensitive operations like RDMA-based gradient synchronization. In AI workloads, where delays or packet loss can slow down or even interrupt training, adaptive routing plays a critical role in maintaining system performance.

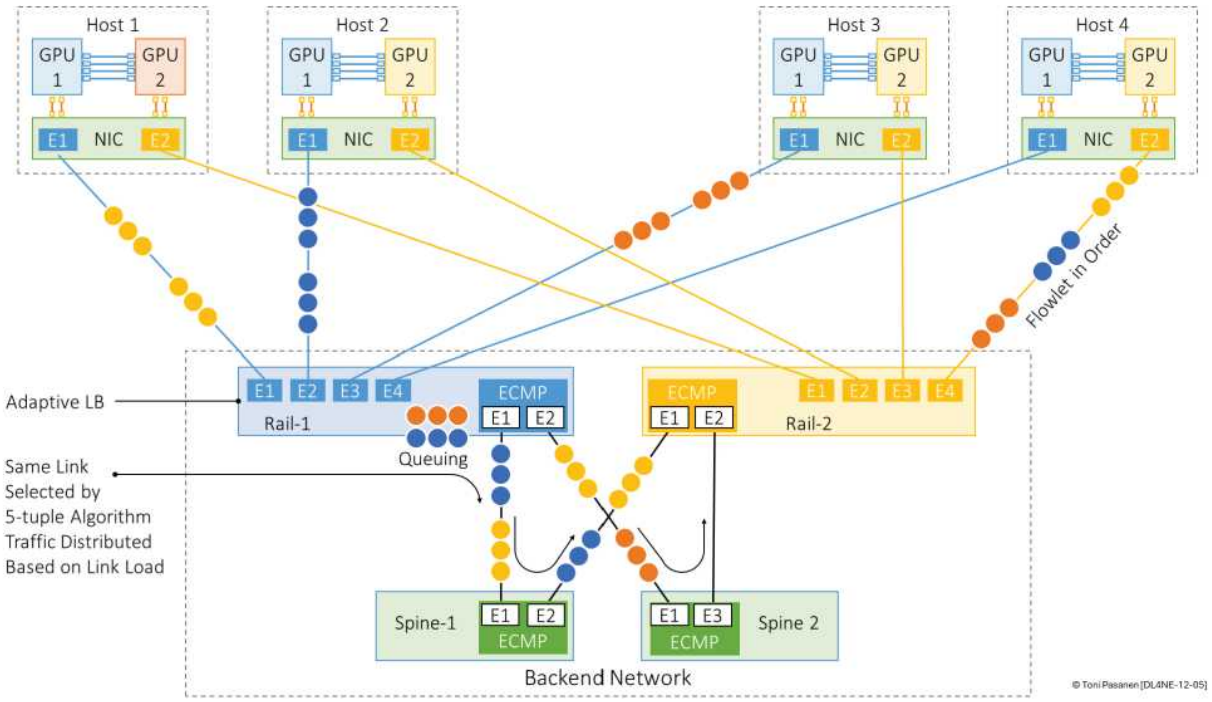


Figure 12-5: *Dynamic Flow Balancing.*

PACKET-BASED LOAD BALANCING WITH PACKET SPRAYING

Packet spraying is a load balancing method where individual packets from the same flow are distributed across multiple equal-cost paths. The idea is to use all available links evenly and reduce the chance of congestion on any single path.

In a RoCEv2-based AI backend network, however, packet spraying presents serious challenges. RoCEv2 relies on lossless and in-order packet delivery. When packets are sprayed over different paths, they can arrive out of order at the destination. This packet reordering can disrupt RDMA operations and reduce the overall performance of GPU-to-GPU communication.

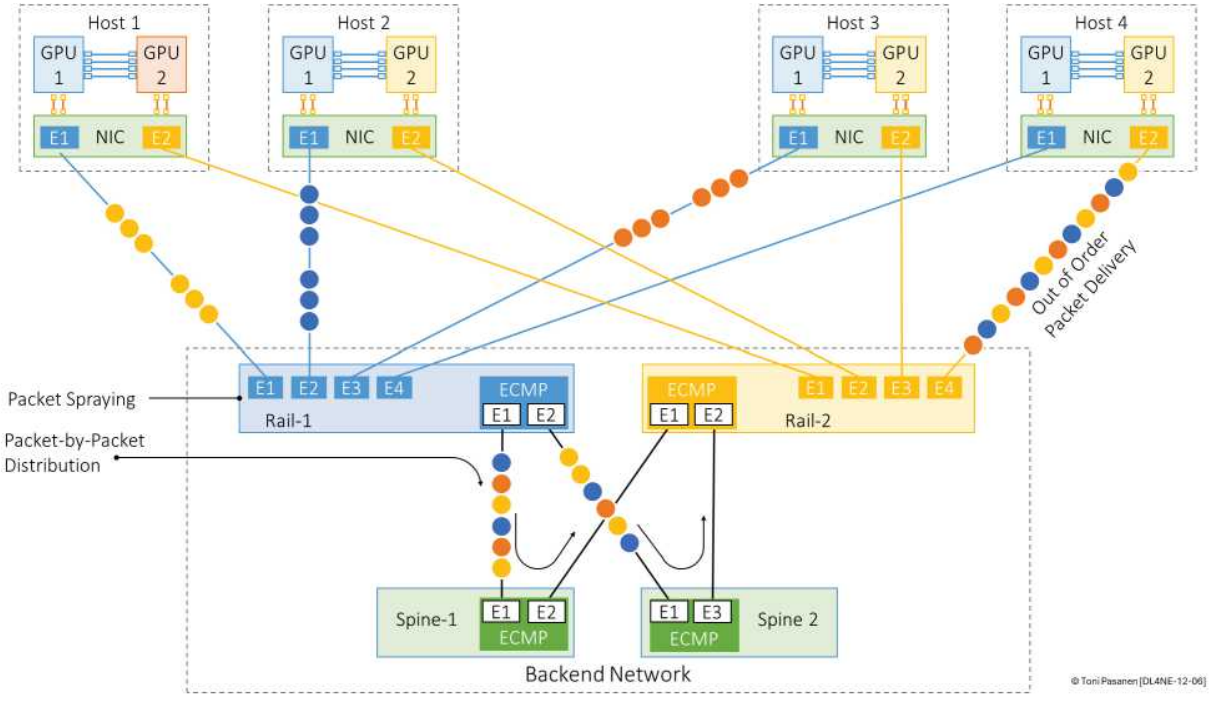
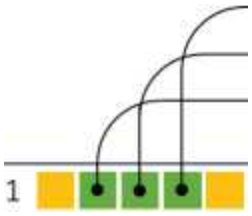


Figure 12-6: *Packet Spraying: OpCode: RDMA Write First, Middle, and Last.*

RDMA Write Only

NVIDIA's RDMA NICs starting from ConnectX-5 support the RDMA Write Only operation, where a RETH header is included in every packet. Figure 12-7 shows how the RDMA NIC uses the OpCode: RDMA Write Only in the IBTH header for each message. With this OpCode, every message also includes a RETH header, which holds information about the destination memory block reserved for the data carried in the payload. This allows the receiving RDMA NIC to write data directly to the correct memory location without relying on prior messages in the transfer sequence.

RDMA Write Only, when combined with Packet-Based Load Balancing using Packet Spraying, brings significant benefits. Since each packet is selfcontained and includes full memory addressing information, the network fabric can forward individual packets over different paths without worrying about packet ordering or context loss. This enables true flowlet or even per-packet load balancing, which helps spread traffic more evenly across available links, avoids hotspots, and reduces queuing delays.



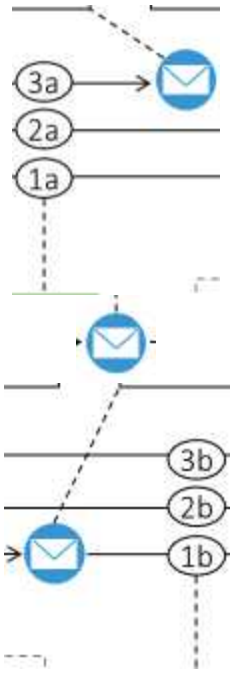


Figure 12-7: *Packet Spraying: OpCode: TDMA Write Only.*

CONFIGURING PER-PACKET LOAD BALANCING ON CISCO NEXUS SWITCHES

At the time of writing, Cisco Nexus 9000 Series Cloud Scale switches (9300-FX3, GX, GX2, and HX-TOR), starting from NX-OS Release 10.5(1)F, support Dynamic Load Balancing (DLB)—including flowlet-based and per-packet (packet spraying) load balancing. DLB is supported on Layer 3 physical interfaces in IP-routed and VXLAN fabrics for unicast IPv4 and IPv6 traffic.

When DLB is enabled, egress QoS and access policies are not applied to flows using DLB. Similarly, TX SPAN configured on an egress interface does not capture DLB traffic. For hardware and software support details, refer to Cisco's official documentation.

Example 12-1 shows a basic configuration for enabling per-packet load balancing:

switch(config)# hardware profile dlb switch(config-dlb)# dlb-interface Eth1/1 switch(config-dlb)# dlb-interface Eth1/2 switch(config-dlb)# mac-address aa:bb:cc:dd:ee:ff switch(config-dlb)# mode per-packet

Example 12-1: *Configuring Per-Packet Load Balancing Packet Spraying*.

Note: The DLB MAC acts as a virtual next-hop MAC address. It's not tied to any specific physical interface. This decouples the

MAC from the physical path, allowing the switch to choose a different egress port for each packet. The same DLB MAC address must be configured on all participating switches. If you do not specify a DLB MAC, the default DLB MAC 00:CC:CC:CC:CC:CC is applied.

REFERENCES

[1] V. Xu, et al., “Fully Adaptive Routing Ethernet”, February 25, 2024, Internet-Draft: draft-xu-lsr-fare-02.

[2] Paul Congdon, Yolanda Yu, “Technologies for the Lossless Network for Data Centers”, March 18, 2018.

[3] Advait Dixit, et al., Purdue University, “On the Impact of Packet Spraying in Data Center Networks”.

[4] Chuhao Chen Fudan University, et al., , et al., “HF²T: Host-Based Flowlet Fine-Tuning for RDMA Load Balancing”.

[5] Cisco Nexus 9000 Series NX-OS Unicast Routing Configuration Guide, Release 10.5(x), https://www.cisco.com/c/en/us/td/docs/dcn/nx-os/nexus9000/105x/unicast-routing-configuration/cisco-nexus-9000-series-nx-os-unicast-routing-configuration-guide/m_managing_the_unicast_rib_and_fib.html

CHAPTER 13: BACKEND NETWORK TOPOLOGIES

INTRODUCTION

Although there are best practices for AI Fabric backend networks, such as Data Center Quantized Congestion Control (DCQCN) for congestion avoidance, rail-optimized routed Clos fabrics, and Layer 2 Rail-Only topologies for small-scale implementations, each vendor offers its own validated design. This approach is beneficial because validated designs are thoroughly tested, and when you build your system based on the vendor's recommendations, you receive full vendor support and avoid having to reinvent the wheel.

However, instead of focusing on any specific vendor's design, this chapter explains general design principles for building a resilient, non-blocking, and lossless Ethernet backend network for AI workloads.

Before diving into backend network design, this chapter first provides a high-level overview of a GPU server based on NVIDIA H100 GPUs. The first section introduces a shared NIC architecture, where 8 GPUs share two NICs. The second section covers an architecture where each of the 8 GPUs has a dedicated NIC.

Shared NIC

Figure 13-1 illustrates a shared NIC approach. In this example setup, NVIDIA H100 GPUs 0–3 are connected to NVSwitch chips 1-1, 1-2, 1-3, and 1-4 on baseboard-1, while GPUs 4–7 are connected to NVSwitch chips 2-1, 2-2, 2-3, and 2-4 on baseboard-2. Each GPU connects to all four NVSwitch chips on its respective baseboard using a total of 18 NVLink 4 connections: 5 links to chip 1-1, 4 links to chip 1-2, 4 links to chip 1-3, and 5 links to chip 1-4.

The NVSwitch chips themselves are paired between the two baseboards. For example, chip 1-1 on baseboard-1 connects to chip 2-1 on baseboard-2 with four NVLink connections, chip 1-2 connects to chip 2-2, and so on. This design forms a fully connected crossbar topology across the entire system.

Thanks to this balanced pairing, GPU-to-GPU communication is very efficient whether the GPUs are located on the same baseboard or on different baseboards. Each GPU can achieve up to 900 GB/s of total GPU-to-GPU bandwidth at full NVLink 4 speed.

For inter-GPU server connection, GPUs are also connected to a shared NVIDIA ConnectX-7 200 GbE NIC through a PEX89144 PCIe Gen5 switch. Each GPU has a dedicated PCIe Gen5 x16 link to the switch, providing up to 64 GB/s of bidirectional bandwidth (32 GB/s in each direction) between the GPU and the switch. The ConnectX-7 (200Gbps) NIC is also connected to the same PCIe switch, enabling high-speed data transfers between remote GPUs and the NIC through the PCIe fabric.

While each GPU benefits from a high-bandwidth, low-latency PCIe connection to the switch, the NIC itself has a maximum network bandwidth of 200 GbE, which corresponds to roughly 25 GB/s. Therefore, the PCIe switch is not a bottleneck; instead, the NIC's available bandwidth must be shared among all eight GPUs. In scenarios where multiple GPUs are sending or receiving data simultaneously, the NIC becomes the limiting factor, and the bandwidth is divided between the GPUs.

In real-world AI workloads, however, GPUs rarely saturate both the PCIe interface and the NIC at the same time. Data transfers between the GPUs and the NIC are often bursty and asynchronous, depending on the training or inference pipeline stage. For example, during deep learning training, large gradients might be exchanged periodically, but not every GPU constantly sends data at full speed. Additionally, many optimizations like gradient compression, pipeline parallelism, and overlapping computation with communication further reduce the likelihood of sustained full-speed congestion.

As a result, even though the NIC bandwidth must be shared, the shared ConnectX-7 design generally provides sufficient network performance for typical AI workloads without significantly impacting training or inference times.

In high-performance environments, such as large-scale training workloads or GPU communication across nodes, this shared setup can become a bottleneck. Latency may increase under load, and data transfer speeds can slow down.

Despite these challenges, the design is still useful in many cases. It is well-suited for development environments, smaller models, or setups where cost is a primary concern. If the workload does not require maximum GPU-to-network performance, sharing a NIC across GPUs can be a reasonable and efficient solution. However, for optimal performance and full support for technologies like GPUDirect RDMA, it is better to use a dedicated NIC for each GPU.

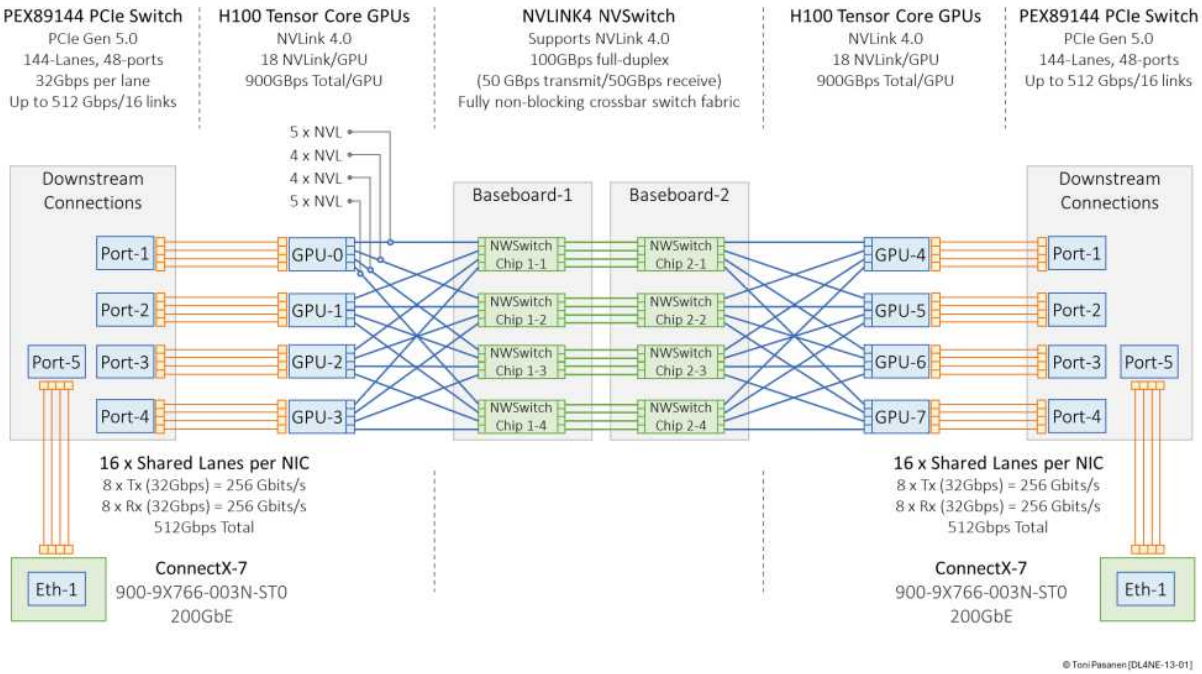


Figure 13-1: *Shared NIC GPU Server.*

NIC per GPU

Figure 13-2 builds on the shared NIC design from Figure 13-1 but takes a different approach. In this setup, each GPU has its own dedicated ConnectX-7 200 GbE NIC. All NICs are connected to the PCIe Gen5 switch, just like in the earlier setup, but now each

GPU uses its own PCIe Gen5 x16 connection to a dedicated NIC. This design eliminates the need for NIC sharing and allows every GPU to use the full 64 GB/s PCIe bandwidth independently.

The biggest advantage of this design is in GPU-to-NIC communication. There is no bandwidth contention at the PCIe level, and each GPU can fully utilize RDMA and GPUDirect features with its own NIC. This setup improves network throughput and reduces latency, especially in multinode training workloads where GPUs frequently send and receive large amounts of data over Ethernet.

The main drawback of this setup is cost. Adding one NIC per GPU increases both hardware costs and power consumption. It also requires more switch ports and cabling, which may affect system design. Still, these trade-offs are often acceptable in performance-critical environments.

This overall design reflects NVIDIA's DGX and HGX architecture, where GPUs are fully interconnected using NVLink and NVSwitch and each GPU is typically paired with a dedicated ConnectX or BlueField NIC to maximize network performance. In addition, this configuration is well suited for rail-optimized backend networks, where consistent per-GPU network bandwidth and predictable east-west traffic patterns are important.

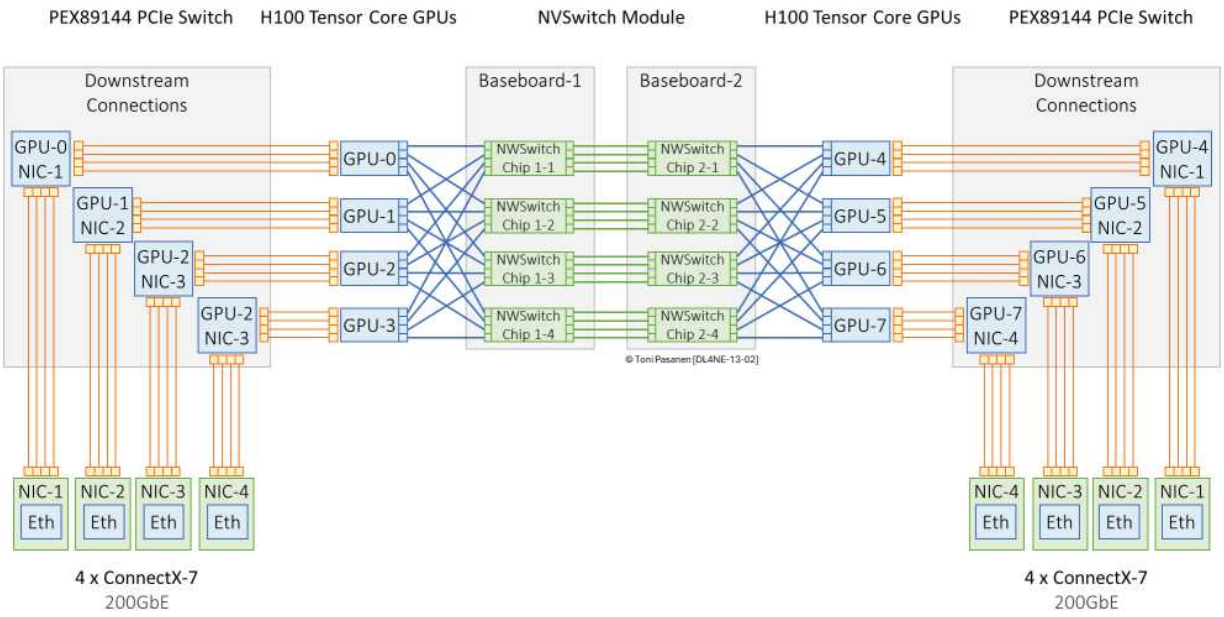


Figure 13-2: *Dedicated NIC per GPU.*

Before moving to the design sections, it is worth mentioning that the need for a high-performance backend network, and how it is designed, is closely related to the size of the neural networks being used. Larger models require more GPU memory and often must be split across multiple GPUs or even servers. This increases the need for fast, low-latency communication between GPUs, which puts more pressure on the backend network.

Figure 13-3 shows a GPU server with 8 GPUs. Each GPU has 80 GB of memory, giving a total of 640 GB GPU memory. This kind of setup is common in high-performance AI clusters.

The figure also shows three examples of running large language models (LLMs) with different parameter sizes:

- **8B model:** This model has 8 billion parameters and needs only approximately 16 GB of memory. It fits on a single

GPU if model parallelism is not required.

- **70B model:** This larger model has 70 billion parameters and needs approximately 140 GB of memory. It cannot fit into one GPU, so it must use at least two GPUs. In this case, the GPUs communicate using intrahost GPU connections across NVLink.

- **405B model:** This large model has 405 billion parameters and needs approximately 810 GB of memory. It does not fit into one server. Running this model requires at least 10 GPUs across multiple servers. The GPUs must use both intra-GPU connections inside a server and inter-GPU connections between servers.

This figure highlights how model size directly affects memory needs, and the number of GPUs required. As models grow, parallelism and fast GPU interconnects become essential.

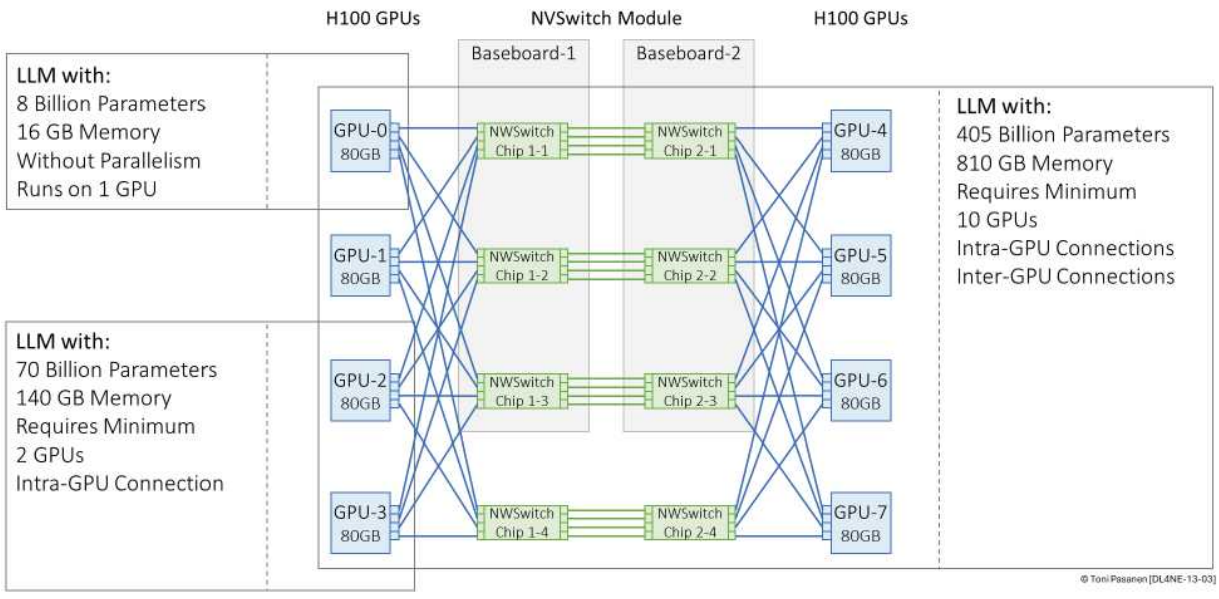


Figure 13-3: *Model Size and Required GPUs.*

DESIGN SCENARIOS

Single Rail Switch Design with Dedicated, Single-Port NICs per GPU

Figure 13-4 illustrates a single rail switch design. The switch interfaces are divided into three groups of eight 200 Gbps interface each. The first group of eight ports is reserved for Host-1, the second group for Host-2, and the third group for Host-3. Each host has eight GPUs, and each GPU is equipped with a dedicated, single-port NIC.

Within each group, ports are assigned to different VLANs to separate traffic into different logical rails. Specifically, the first port of each group belongs to the VLAN representing Rail-1, the second port belongs to Rail-2, and so on. This pattern continues across all three host groups.

Benefits

- Simplicity: The architecture is very easy to design, configure, and troubleshoot. A single switch and straightforward VLAN assignment simplify management.
- Cost-Effectiveness: Only one switch is needed, reducing capital expenditure (CapEx) compared to dual-rail or redundant designs. Less hardware also means lower

operational expenditure (OpEx), including reduced power, cooling, and maintenance costs. Additionally, fewer devices translate to lower subscription-based licensing fees and service contract costs, further improving the total cost of ownership.

- Efficient Use of Resources: Ports are used efficiently by directly mapping each GPU's NIC to a specific port on the switch, minimizing wasted capacity.
- Low Latency within the Rail: Since all communications stay within the same switch, latency is minimized, benefiting tightly-coupled GPU workloads.
- Sufficient for Smaller Deployments: In smaller clusters or test environments where absolute redundancy is not critical, this design is perfectly sufficient.

Drawbacks

- No Redundancy: A single switch creates a single point of failure. If the switch fails, all GPU communications are lost.
- Limited Scalability: Expanding beyond the available switch ports can be challenging. Adding more hosts or GPUs might require replacing the switch or redesigning the network.
- Potential Oversubscription: With all GPUs sending and receiving traffic through the same switch, there's a risk of oversubscription, especially under heavy AI workload patterns where network traffic bursts are common.

- **Difficult Maintenance:** Software upgrades or hardware maintenance on the switch impact all connected hosts, making planned downtime more disruptive.
- **Not Suitable for High Availability (HA) Requirements:** Critical AI workloads, especially in production environments, often require dualrail (redundant) networking to meet high availability requirements. This design would not meet such standards.

Single rail designs are cost-efficient and simple but lack redundancy and scalability, making them best suited for small or non-critical AI deployments.

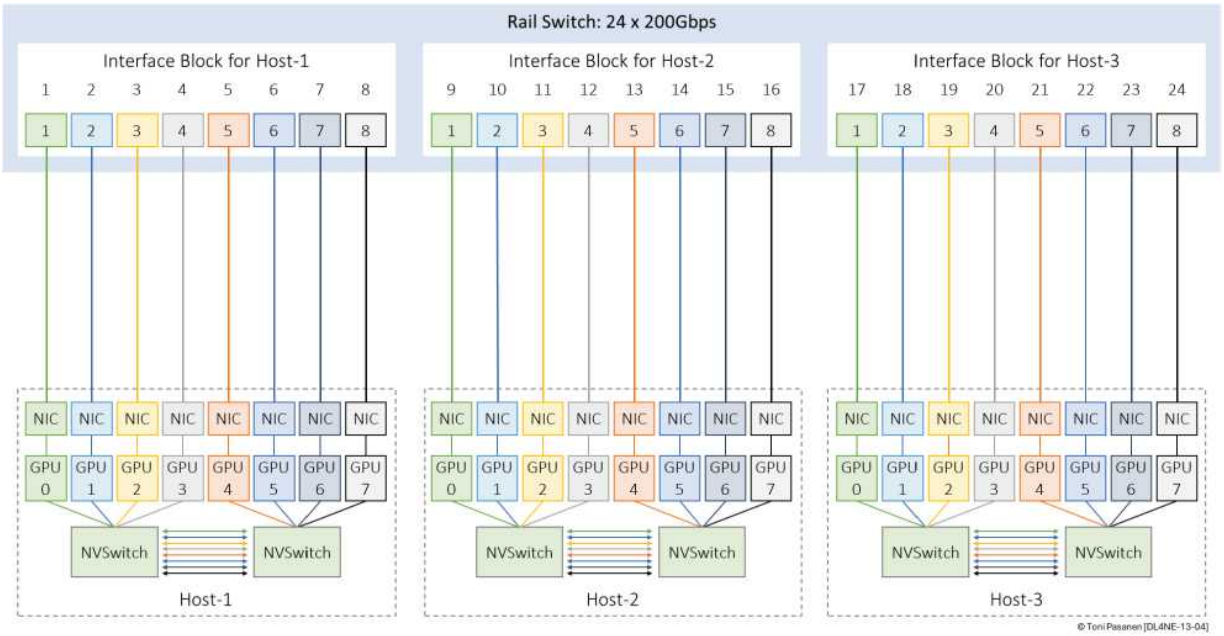


Figure 13-4: *Single Rail Switch Design: GPU with Single Port NIC.*

Dual-Rail Switch Topology with Dedicated, Dual-Port NICs per GPU

In this topology, each host contains 8 GPUs, and each GPU has a dedicated dual-port NIC. The NICs are connected across two independent Rail switches equipped with 200 Gbps interfaces. This design ensures that every GPU has redundant network connectivity through separate switches, maximizing performance, resiliency, and failover capabilities.

Each Rail switch independently connects to one port of each NIC, creating a dual-homed connection per GPU. To ensure seamless operations and redundancy, the two switches must logically appear as a single device to the host NICs, even though they are physically distinct systems.

Benefits

- High Availability: The failure of a single switch, link, or NIC port does not isolate any GPU, maintaining system uptime.
- Load Balancing: Traffic can be distributed across both switches, maximizing bandwidth utilization and reducing bottlenecks.
- Scalability: Dual-rail architectures can be extended easily to larger deployments while maintaining predictable performance and redundancy.
- Operational Flexibility: Maintenance can often be performed on one switch without service disruption.

Drawbacks

- **Higher Cost:** Requires two switches, twice the number of cables, and dual-port NICs, increasing CapEx and OpEx.
- **Complexity:** Managing a dual-rail environment introduces more design complexity due to Multi-Chassis Link Aggregation (MLAG).
- **Increased Power and Space Requirements:** Two switches and more cabling demand more rack space, power, and cooling.

Challenges of Multi-Chassis Link Aggregation (MLAG)

To create a logical channel between dual-port NICs and two switches, the switches must be presented as a single logical device to each NIC. **MultiChassis Link Aggregation (MLAG)** is often used for this purpose. MLAG allows a host to see both switch uplinks as part of the same LAG (Link Aggregation Group).

Another solution is to assign the two NIC ports to different VLANs without bundling them into a LAG, though this approach may limit bandwidth utilization and redundancy benefits compared to MLAG.

MLAG introduces several challenges:

- **MAC Address Synchronization:** Both switches must advertise the same MAC address to the host NICs, allowing the two switches to appear as a single device.

- Port Identification: A common approach to building MLAG is to use the same interface numbers on both switches. Therefore, the system must be capable of uniquely identifying each member link internally.
- Control Plane Synchronization: The two switches must exchange state information (e.g., MAC learning, link status) to maintain a consistent and synchronized view of the network.
- Failover Handling: The switches must detect failures quickly and handle them gracefully without disrupting existing sessions, requiring robust failure detection and recovery mechanisms.

Vendor-Specific MLAG Solutions

The following list shows some of the vendor proprietary MLAG:

- Cisco Virtual Port Channel (vPC): Cisco's vPC allows two Nexus switches to appear as one logical switch to connected devices, synchronizing MAC addresses and forwarding state.
- Juniper Virtual Chassis / MC-LAG: Juniper offers Virtual Chassis and MC-LAG solutions, where two or more switches operate with a shared control plane, presenting themselves as a single switch to the host.

- Arista MLAG: Arista Networks implements MLAG with a simple peerlink architecture, supporting independent control planes while synchronizing forwarding state.
- NVIDIA/Mellanox MLAG: Mellanox switches also offer MLAG solutions, often optimized for HPC and AI workloads.

Standards-Based Alternative: EVPN ESI Multihoming

Instead of vendor-specific MLAG, a standards-based approach using Ethernet Segment Identifier (ESI) Multihoming under BGP EVPN can be used. In this model:

- Switches advertise shared Ethernet segments (ESIs) to the host over BGP EVPN.
- Hosts see multiple physical links but treat them as part of a logical redundant connection.
- EVPN ESI Multihoming allows for interoperable solutions across vendors, but typically adds more complexity to the control plane compared to simple MLAG setups.

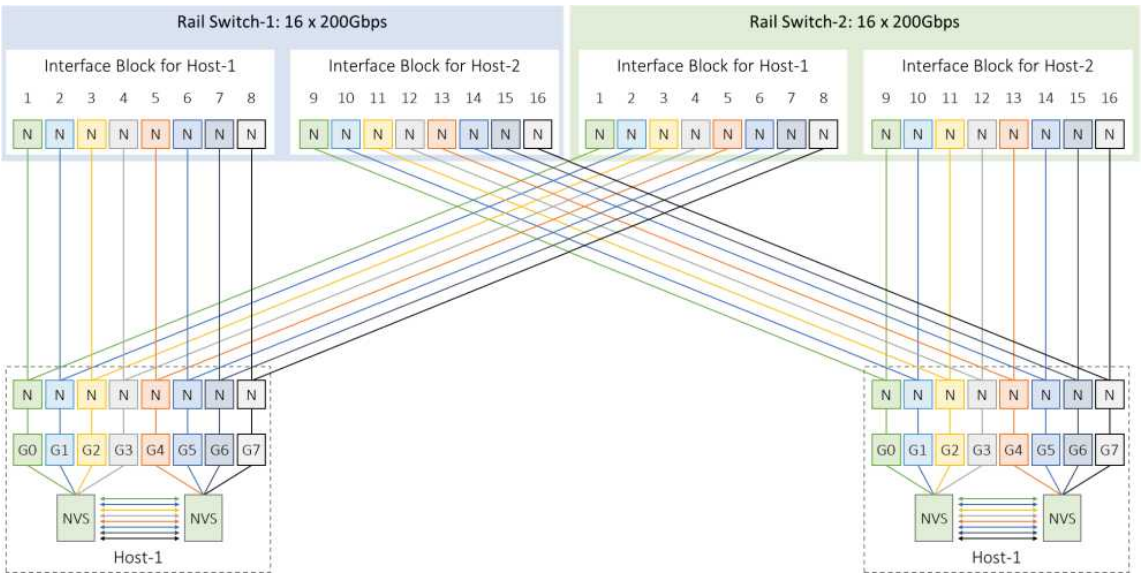


Figure 13-5: *Dual Rail Switch Design: GPU with Dual-Port NIC.*

Cross-Rail Communication over NVLink in Rail-Only Topologies

In the introduced single- and dual-rail topologies (Figures 13-4 and 13-5), each GPU is connected to a dedicated NIC, and each NIC connects to a specific Rail switch. However, there is no direct cross-rail connection between the switches themselves — no additional spine layer interconnecting the rails. As a result, if a GPU needs to send data to a destination GPU that belongs to a different rail, special handling is required within the host before the data can exit over the network.

For example, consider a memory copy operation where GPU-2 (connected to Rail 3) on Host-1 needs to send data to GPU-3 (connected to Rail 4) on Host-2. Since GPU-2's NIC is associated with Rail 3 and GPU-3 expects data arriving over Rail 4, the communication path must traverse multiple stages:

1. Intra-Host Transfer: The data is first copied locally over NVLink from GPU-2 to GPU-3 within Host-1. NVLink provides a high-bandwidth, low-latency connection between GPUs inside the same server.

2. NIC Transmission: Once the data resides in GPU-3's memory, it can be sent out through GPU-3's NIC, which connects to Rail 4.

3. Inter-Host Transfer: The packet travels over Rail 4 through one of the Rail switches to reach Host-2.

4. Destination Reception: Finally, the data is delivered to GPU-3 on Host-2.

This method ensures that each network link (and corresponding NIC) is used according to its assigned rail without needing direct switch-to-switch rail interconnects.

To coordinate and optimize such multi-step communication, NVIDIA Collective Communications Library (NCCL) plays a critical role. NCCL automatically handles GPU-to-GPU communication across multiple nodes and rails, selecting the appropriate path, initiating memory copies over NVLink, and scheduling transmissions over the correct NICs — all while maximizing bandwidth and minimizing latency. The upcoming chapter will explore NCCL in greater detail.

Figure 13-6 illustrates how the upcoming topology in Figure 13-7 maps NIC-to-Rail connections, transitioning from a switch interface-based view to a rail-based view. Figure 13-6 shows a

partial interface layout of a Cisco Nexus 9348D-GX2A switch and how its interfaces are grouped into different rails as follows:

- Rail-1 Interfaces: 1, 4, 7, 10
- Rail-2 Interfaces: 13, 16, 19, 22
- Rail-3 Interfaces: 25, 28, 31, 34
- Rail-4 Interfaces: 37, 40, 43, 46
- Rail-5 Interfaces: 2, 5, 8, 11
- Rail-6 Interfaces: 14, 17, 20, 23
- Rail-7 Interfaces: 26, 29, 32, 35
- Rail-8 Interfaces: 38, 41, 44, 47

However, a port-based layout becomes extremely messy when describing larger implementations. Therefore, the common practice is to reference the rail number instead of individual switch interface identifiers.

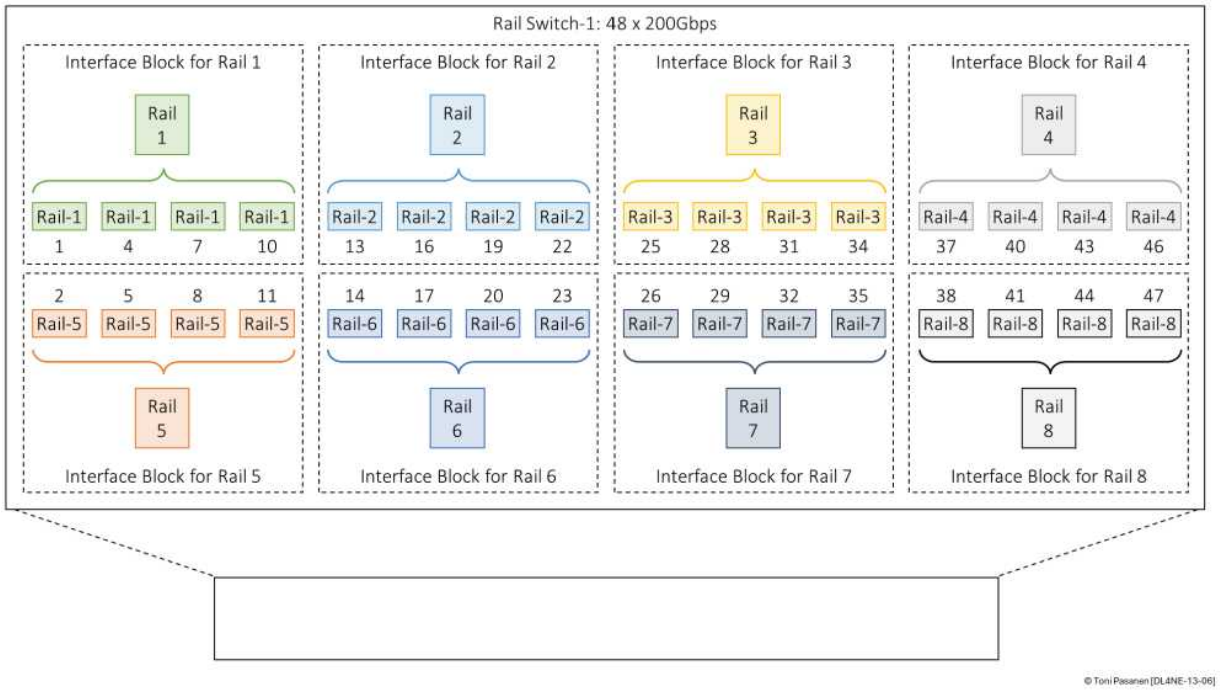


Figure 13-6: *Interface Block to Rail Mapping*.

Figure 13-7 provides an example showing how each NIC is now connected to a rail instead of being directly mapped to a specific physical interface. In this approach, each rail represents a logical group of physical interfaces, simplifying the overall design and making larger deployments easier to visualize and document.

In our example "Host-Segment" (an unofficial name), we have four hosts, each equipped with eight GPUs — 32 GPUs in total. Each GPU has a dedicated 200 Gbps dual-port NIC. All GPUs are connected to two rail switches over a 2 × 200 Gbps MLAG, providing 400 Gbps of transmission speed per GPU.

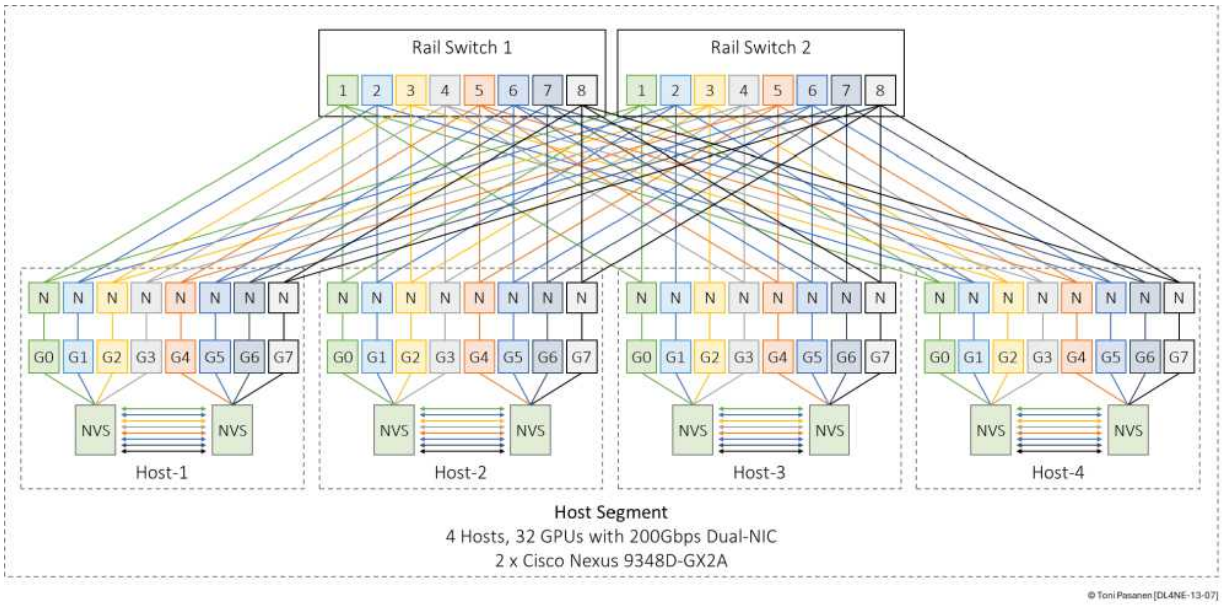


Figure 13-7: *Example Figure of Connecting 32 Dual-Port NICs 8 Rails on 2 Switches.*

Figure 13-8 shows how multiple Host-Segments can be connected. The figure illustrates a simplified two-tier, three-stage Clos fabric topology, where full-mesh Layer 3 links are established between the four Rail switches (leaf switches) and the Spine switches. The figure also presents the link capacity calculations. Each Rail switch has 32 × 100 Gbps connections to the hosts, providing a total downlink capacity of 3.2 Tbps.

Since oversubscription is generally not preferred in GPU clusters — to maintain high performance and low latency — the uplink capacity from each Rail switch to the Spine layer must also match 3.2 Tbps. To achieve this, each Rail switch must have uplinks capable of an aggregate transfer rate of 3.2 Tbps. This can be implemented either by using native 800 Gbps interfaces or by forming a logical Layer 3 port channel composed of two 400 Gbps links per Spine connection. Additionally, Inter-Switch

capacity can be increased by adding more switches in the Spine layer. This is one of the benefits of a Clos fabric: the capacity can be scaled without the need to replace 400 Gbps interfaces with 800 Gbps interfaces, for example.

This topology forms a Pod and supports 64 GPUs in total and provides a non-blocking architecture, ensuring optimal east-west traffic performance between GPUs across different Host-Segments.

In network design, the terms "two-tier" and "three-stage" Clos fabric describe different aspects of the same overall topology. "Two-tier" focuses on the physical switch layers (typically Leaf and Spine) and describes the depth of the topology, offering a hierarchy view of the architecture. Essentially, it's concerned with how many switching layers are present. On the other hand, three-stage Clos describes the logical data path a packet follows when moving between endpoints: Leaf–Spine–Leaf. It focuses on how data moves through the network and the stages traffic flows through. Therefore, while a two-tier topology refers to the physical switch structure, a three-stage Clos describes the logical path taken by packets, which crosses through three stages: Leaf, Spine, and Leaf. These two perspectives are complementary, not contradictory, and together they provide a complete view of the Clos network design.

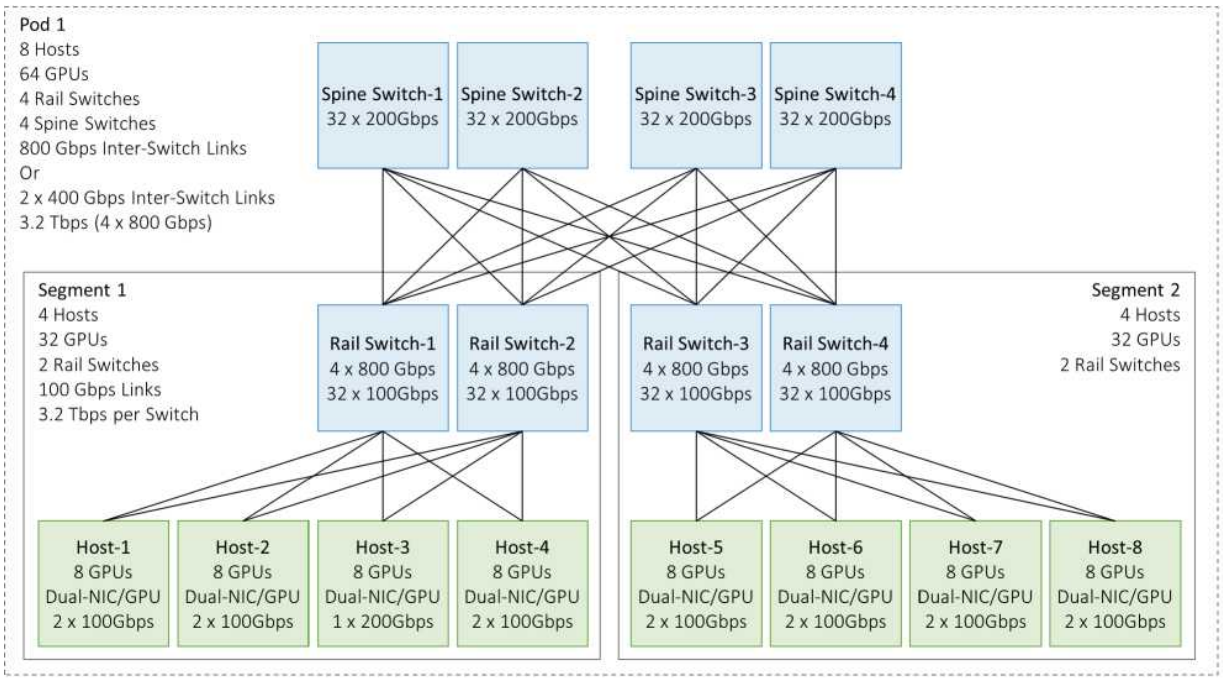


Figure 13-8: *AI fabric – Pod Design.*

Figure 13-9 extends the previous example by adding a second 64-GPU Pod, creating a larger multi-Pod architecture. To interconnect the two Pods, four Super-Spine switches are introduced, forming an additional aggregation layer above the Spine layer. Each Pod retains its internal two-tier Clos fabric structure, with Rail switches fully meshed to the Spine switches as described earlier. The Spine switches from both Pods are then connected northbound to the Super-Spine switches over Layer 3 links.

Due to the introduction of the Super-Spine layer, the complete system now forms a three-tier, five-stage Clos topology. This design supports scalable expansion while maintaining predictable latency and high bandwidth between GPUs across different Pods. Like the Rail-to-Spine design, maintaining a non-blocking architecture between the Spine and SuperSpine layers

is critical. Each Spine switch aggregates 3.2 Tbps of traffic from its Rail switches; therefore, the uplink capacity from each Spine to the Super-Spine layer must also be 3.2 Tbps.

This can be achieved either by using native 800 Gbps links or logical Layer 3 port channels composed of two 400 Gbps links per Super-Spine connection. All Spine switches are fully meshed with all Super-Spine switches to ensure high availability and consistent bandwidth. This architecture enables seamless east-west traffic between GPUs located in different Pods, ensuring that inter-Pod communication maintains the same non-blocking performance as intra-Pod traffic.

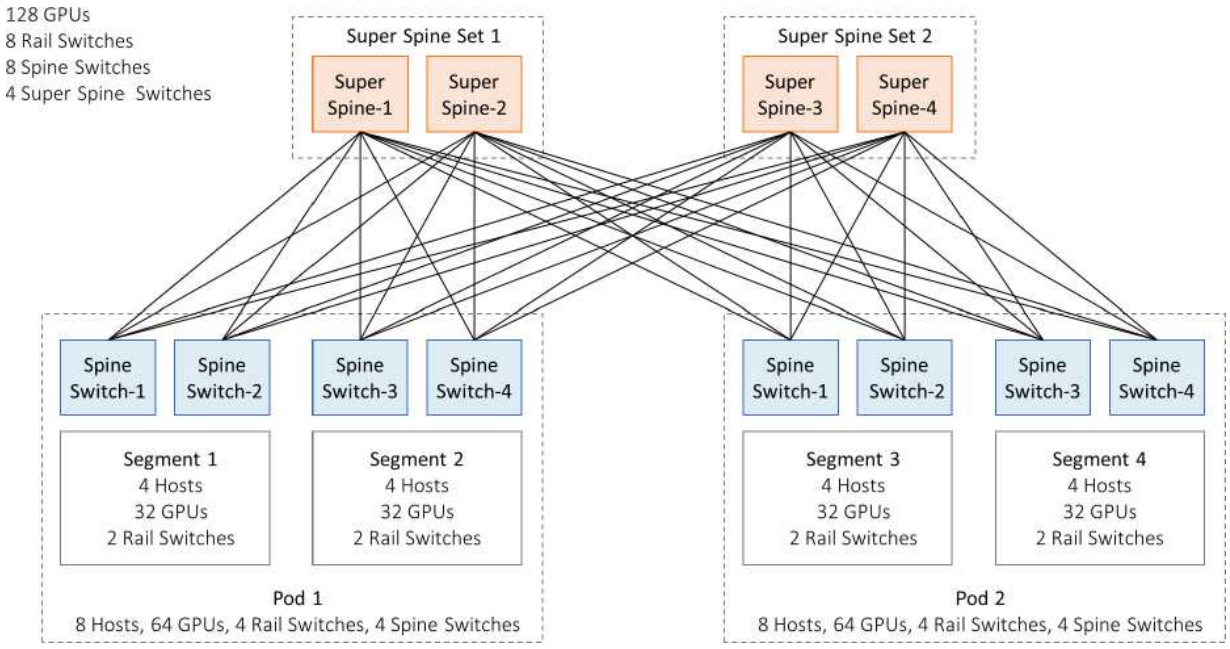


Figure 13-9: *AI fabric – Multi-Pod Design.*

Rail Desings in GPU Fabric

When building a scalable, resilient GPU network fabric, the design of the rail layer, the portion of the topology that

interconnects GPU servers via Top-of-Rack (ToR) switches, plays a critical role. This section explores three different models: Multi-rail-per-switch, Dual-rail-per-switch, and Single-rail-per-switch. All three support dual-NIC-per-GPU designs, allowing each GPU to connect redundantly to two separate switches, thereby removing the Rail switch as a single point of failure.

Multi-Rail-per-Switch

In this model, multiple small subnets and VLANs are configured per switch, with each logical rail mapped to a subset of physical interfaces. For example, a single 48-port switch might host four or eight logical rails using distinct Layer 2 and Layer 3 domains. Because all logical rails share the same physical device, isolation is logical. As a result, a hardware or software failure in the switch can impact all rails and their associated GPUs, creating a large failure domain. This model is not part of NVIDIA's validated Scalable Unit (SU) architecture but may suit test environments,

development clusters, or small-scale GPU fabrics where hardware cost efficiency is a higher priority than strict fault isolation. From a CapEx perspective, multi-rail-per-switch is the most economical, requiring fewer switches.

Figure 13-10 illustrates the multi-rail-per-switch architecture, where each rail is implemented as a separate VLAN-subnet pair mapped to a subset of switch ports. In the figure, interfaces 1–4 are assigned to subnet 10.0.1.0/28 and VLAN 101, while interfaces 5–8 are mapped to subnet 10.0.2.0/28 and VLAN 102.

Each VLAN maintains its own MAC address table, learning GPU NIC MACs through ingress traffic. Although not shown in the figure, the Rail switch acts as the default gateway for all eight VLANs. The figure also illustrates the BGP process when a Clos architecture with a spine layer is used to connect rail switches. All directly connected subnets are installed into the local Routing Information Base (RIB) as connected routes. These routes are then imported into the BGP Loc-RIB. Next, the routes pass through the BGP output policy engine, where they are aggregated into a single summary route: 10.0.1.0/24. This aggregate is placed into the BGP Adj-RIB-Out. When the BGP Update message is sent to a peer, the NEXT_HOP attribute is set accordingly.

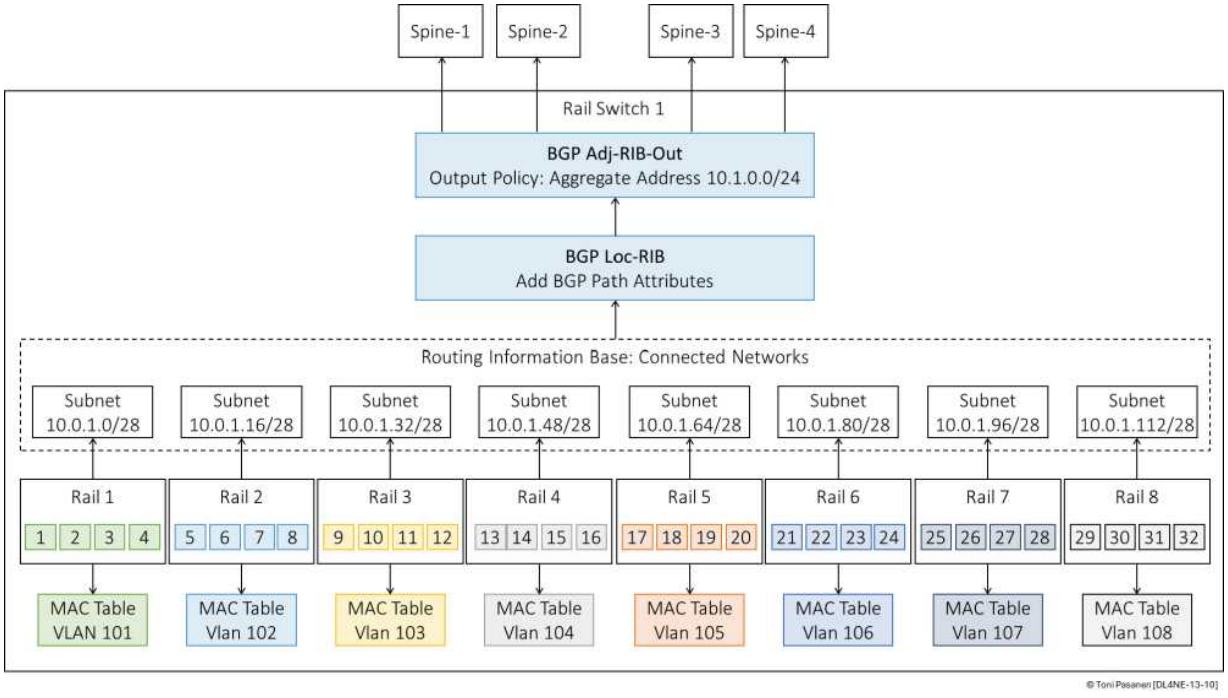


Figure 13-10: *Multi-Rail per Switch.*

Dual-Rail-per-Switch

While dual-rail-per-switch improves manageability and is easier to scale, it shares the same limitation: both logical rails reside within a single physical switch, so the failure domain remains large. A single switch failure or misconfiguration affects both rails and all associated GPUs.

This design resembles the dual-rail concept used in scalable AI clusters, but NVIDIA's SU approach calls for two separate physical switches per rail, which provides full physical isolation. Dual-rail-per-switch hits a middle ground in terms of CapEx and OpEx: fewer switches are required than in the single-rail model, and operational complexity is reduced compared to multi-rail. It's often a good choice for intermediate-scale environments where some fault tolerance and cost control must be balanced.

Figure 13-11 illustrates a dual-rail-per-switch design, where the switch interfaces are divided evenly between two separate rails. Rail 1 uses interfaces 1 through 16 and is assigned to subnet 10.0.1.0/25 (VLAN 101). Rail 2 uses interfaces 17 through 32 and is assigned to subnet 10.0.128.0/25 (VLAN 102). Each VLAN has its own MAC address table, and the rail switch serves as the default gateway for both. The individual /25 subnets are redistributed into the BGP process and summarized as 10.0.1.0/24 for advertisement toward the spine layer.

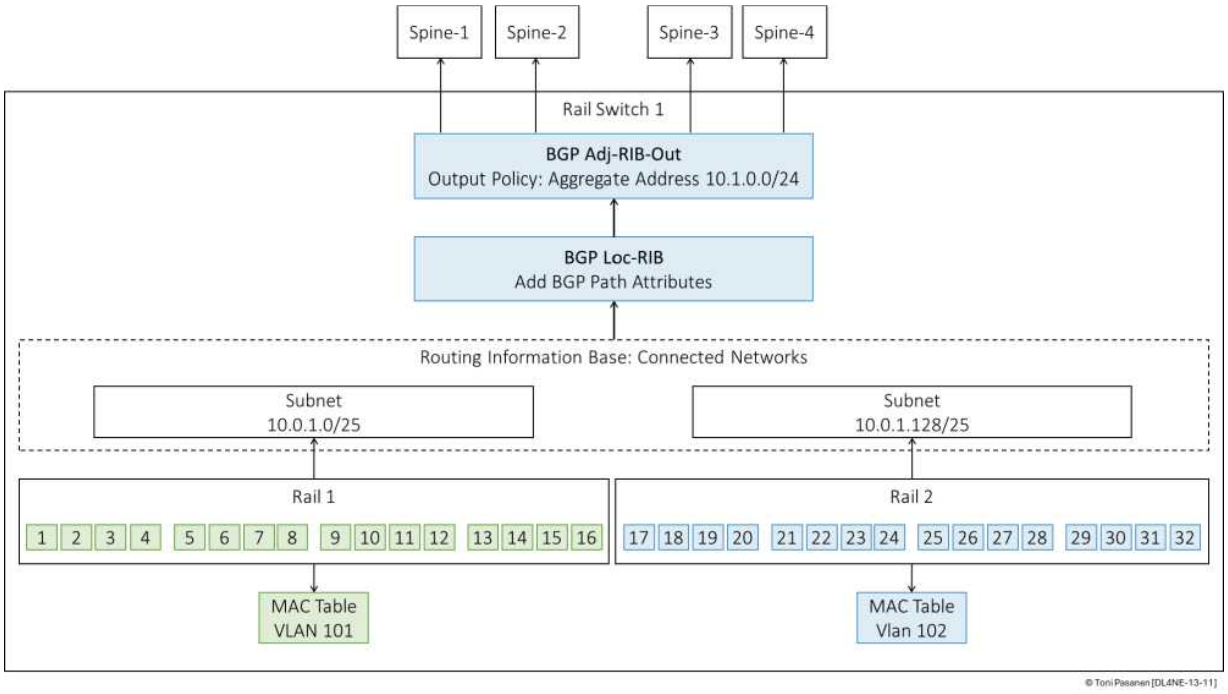


Figure 13-11: *Dual-Rail Switch*.

Single-Rail-per-Switch

This model offers the highest level of physical isolation. Each switch forms a single rail, serving its connected GPU servers through one subnet and one VLAN. No logical separation is needed, as each rail is entirely independent in hardware. As a result, a switch failure affects only the GPU servers attached to that specific rail, yielding a small, predictable failure domain.

The design closely aligns with NVIDIA's Scalable Unit (SU) architecture, in which each rack or rack group includes its own rail switch, and horizontal scaling is achieved by repeating modular, self-contained units.

While this model demands the highest CapEx, due to the one-to-one mapping between switches and rails, it offers major

operational advantages. Configuration is simpler, troubleshooting is faster, and the risk of cascading faults is minimized. There is no need for route summarization, or custom BGP redistribution logic. Over time, these benefits help drive down OpEx, particularly in large-scale or mission-critical GPU clusters.

To ensure optimal hardware utilization, it is important to align the number of GPU servers per rack with the switch's port capacity. Otherwise, underutilized ports can lead to inefficiencies in infrastructure cost and resource planning.

Figure 13-12 illustrates a simplified single-rail-per-switch topology. All interfaces from 1 to 32 operate within a single rail, configured with subnet 10.0.1.0/24 and VLAN 101. The rail switch serves as the default gateway, and because the full /24 subnet is used without subnetting, route summarization is not needed.

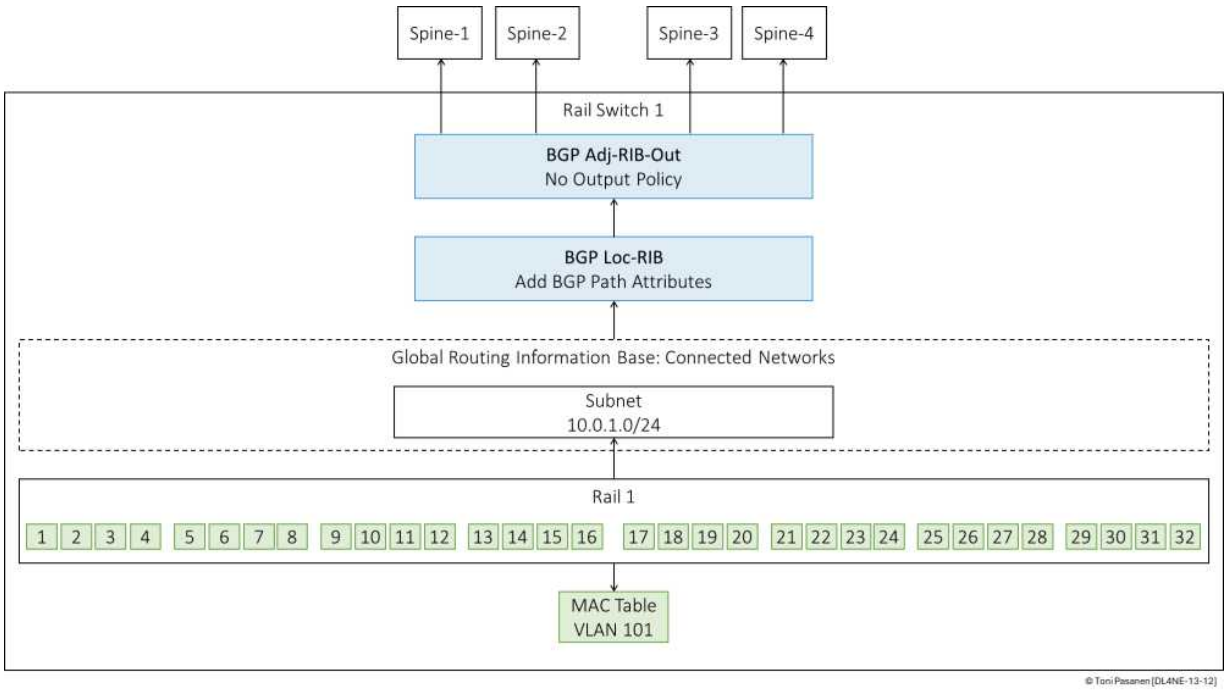


Figure 13-12: *Single-Rail Switch*.

AI Fabric Architecture Conclusion

Figure 13-13 illustrates one way to describe the overall architecture of an AI Fabric. It is divided into three domains. The first domain, called the *Segment*, includes GPU hosts and Rail switches. The second domain, the *Pod*, aggregates multiple segments using Spine switches. In cases where NCCL builds a topology where cross-rail inter-host traffic is first copied to

the local GPU memory (located on the destination rail) and then sent over the GPU NIC to the remote GPU via the correct Rail switch, a Pod architecture with Spine switches may not be necessary. The third domain, *multi-Pod*, interconnects multiple pods using Super Spine switches, enabling large-scale AI Fabric deployments. Figure 13-10 also depicts global settings and properties shared across the AI Fabric backend network.

Segment: GPU I/O Topology and Rail Switch Fabric Profile

GPU I/O Topology: Each GPU connects to the network through a NIC. You can either dedicate a NIC to each GPU or share one NIC among multiple GPUs. NICs may have single, dual, or quad ports and support speeds such as 100, 200, or 400 Gbps. The interconnect type can be InfiniBand, RoCEv2, or NVLink. A segment typically includes multiple hosts.

Rail Switch Fabric Profile: Rail switches connect directly to GPU hosts. Each rail handles a group of NIC ports. You can map rails

one-to-one to switches for physical isolation or map multiple rails per switch for logical isolation. In the latter case, two or more rails can be mapped per switch depending on performance and capacity requirements. Rail switches are responsible for ingress packet classification and for mapping RoCEv2 traffic to the correct queues.

Pod: Spine Switch Profile:

Spine switches aggregate multiple Rail switches, forming a Pod that consists of *n* segments. Spine switches enable cross-rail communication between GPUs. They use high-density, high-speed ports. When the Spine layer is used, the result is a 2-tier, 3-stage architecture.

Multi-Pod: Super Spine Switch Profile

Super Spine switches provide inter-Pod connectivity. They are built with very high port density to support all connected Spine switches. When the Super Spine layer is used, the architecture becomes a 3-tier, 5-stage fabric.

Global AI Fabric Profile

All layers are governed by the Global AI Fabric Profile. This profile defines the control plane (eBGP, iBGP, BGP EVPN), the data plane (Ethernet, VXLAN), Layer 3 ECMP strategies (flow-based, flowlet-based, or perpacket), congestion control

mechanisms (ECN marking, PFC), inter-switch link monitoring (BFD), and global MTU settings.

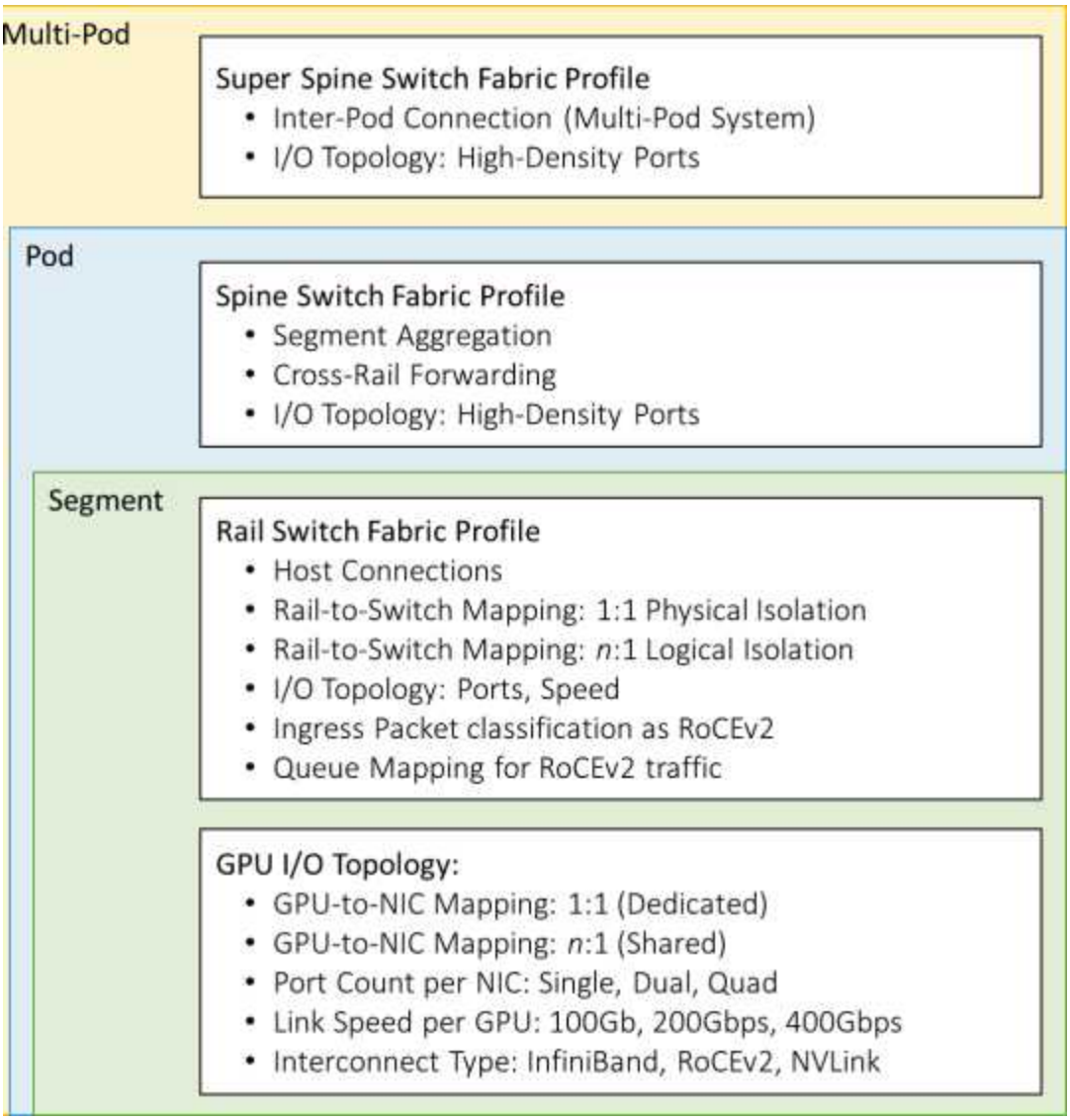


Figure 13-13: *AI fabric Architecture Description.*

Hash Polarization

In 2-tier or 3-tier topologies, there is an inherent risk of hash polarization. Consider a fully connected Clos topology with four leaf switches and four spine switches. Each leaf connects to all

spine switches and uses ECMP (Equal-Cost Multi-Path) to distribute traffic across these equal-cost paths. When each leaf receives a flow from its locally connected host, and all flows are destined for the same endpoint, the ECMP hash function on each leaf determines which spine switch to use as the next hop.

If the ECMP hash function on all leaf switches produces the same result — for example, all leafs selecting the same spine switch as the next hop — then that spine becomes a bottleneck. It receives all flows and must forward them toward the destination, typically through the same inter-switch link. This can lead to buffer overflows or congestion on that path.

This scenario, where multiple flows are consistently mapped to the same next-hop due to deterministic hash outcomes and similar flow characteristics, is known as hash polarization. It often arises when the entropy in the fields used for ECMP hashing (e.g., source and destination IP addresses and ports) is low. When flows have similar headers and the hash function lacks variability, traffic is not evenly spread across available paths.

Hash polarization can be mitigated in several ways. These include increasing entropy in the hash inputs, tuning the hash algorithm to better distribute flows, or using adaptive techniques like flowlet-based load balancing or congestion-aware rehashing. However, one effective method to eliminate hash polarization is through topological design — specifically, by altering the cabling pattern between leaf and spine switches.

Even if ECMP hashing is deterministic, it is possible to physically design the topology so that not all leaf switches are connected to the same subset of spines. In a traditional Clos network, every leaf connects to every spine. If each leaf applies the same hash function and flow headers are similar,

traffic can converge on the same spine — causing the exact polarization problem described earlier.

To address this, one can build a partially connected fabric where each leaf connects to a distinct subset of spines, and each spine serves a subset of leafs. This breaks the symmetry required for hash polarization to occur. Even if all leaf switches produce the same hash result, the next-hop cannot be the same for every leaf — because that spine may not be reachable by all of them. This intentional topological asymmetry ensures that traffic is distributed across multiple spine switches regardless of hash outcomes.

Such structured cabling approaches often rely on deterministic design patterns and can scale efficiently while reducing the risk of bottlenecks caused by hash polarization. The result is improved load distribution without relying solely on hash tuning or adaptive forwarding mechanisms.

REFERENCES

[1] NVIDIA DGX SuperPOD: Scalable Infrastructure for AI Leadership Reference Architecture, October 2021

[2] Kun Qian – Alibaba Cloud, et al., “Alibaba HPN: A Data Center Network for Large Language Model Training”. ACM SIGCOMM, August 4-8, 2024, Sydney, NSW, Australia

[3] Cisco Validated Design for Data Center Networking Blueprint for AI/ML Applications. March 29, 2024.

https://www.cisco.com/c/en/us/td/docs/dcn/whitepapers/cisco-data-center-networking-blueprint-for-ai-ml-applications.html

[4] Aninda Chatterjee, Vivek – Juniper Networks, Designing Data Centers for AI Clusters

https://www.juniper.net/documentation/us/en/software/nce/ai-clusters-data-center-design/ai-clusters-data-center-design.pdf

[5] Arista, “AI Networking”.

https://www.arista.com/assets/data/pdf/Whitepapers/AI-Network-WP.pdf

CHAPTER 14: GPU CLUSTER COMMUNICATION MODEL

The focus of this chapter is to describe what is needed to start a training job and give an overview what happens during the job. In that sense, this is a kind of closing chapter for our “Deep Learning in AI DC journey”. Figure 14-1 shows a high-level, yet simplified architecture and the main building blocks of two GPU hosts, each with four GPUs and their external network connections. Both hosts are part of the same training cluster. As a prerequisite in our example, the following software packages are installed on both hosts:

PyTorch with CUDA and NCCL Support: PyTorch is a deep learning framework that manages the entire training workflow, including data loading, model definition, parallel execution, and gradient synchronization. It defines the structure of the neural

network, such as the number of layers, neurons per layer, activation functions, and initializes the weights automatically.

CUDA (Compute Unified Device Architecture): CUDA is a parallel computing platform and API model from NVIDIA. PyTorch uses CUDA for memory allocation for the data you move from CPU memory (DRAM) to GPU memory (VRAM), including batch tensors, model weights, and intermediate variables during the forward/backward passes. In the forward pass, CUDA does matrix multiplication, computes neuron output values y using the activation functions (e.g., ReLU, Tanh, Sigmoid) and computes the model error. CUDA also handles the backward pass, where it calculates gradients needed for weight updates and updates local weights.

NCCL (NVIDIA Collective Communication Library): NCCL is a multiGPU, topology-aware collective algorithm designed for high-performance data exchange between GPUs, especially in multi-GPU and multi-node systems. It is used during the gradient communication phase of training, where computed gradients are exchanged between GPUs. For example, after CUDA computes gradients on each GPU, NCCL transfers those gradients to other GPUs using collective communication operations. At the time of writing, NCCL supports the following operations: AllReduce, Broadcast, Reduce, AllGather, and ReduceScatter.

PyTorch, CUDA, and NCCL processes and dependencies are explained detail in upcoming section.

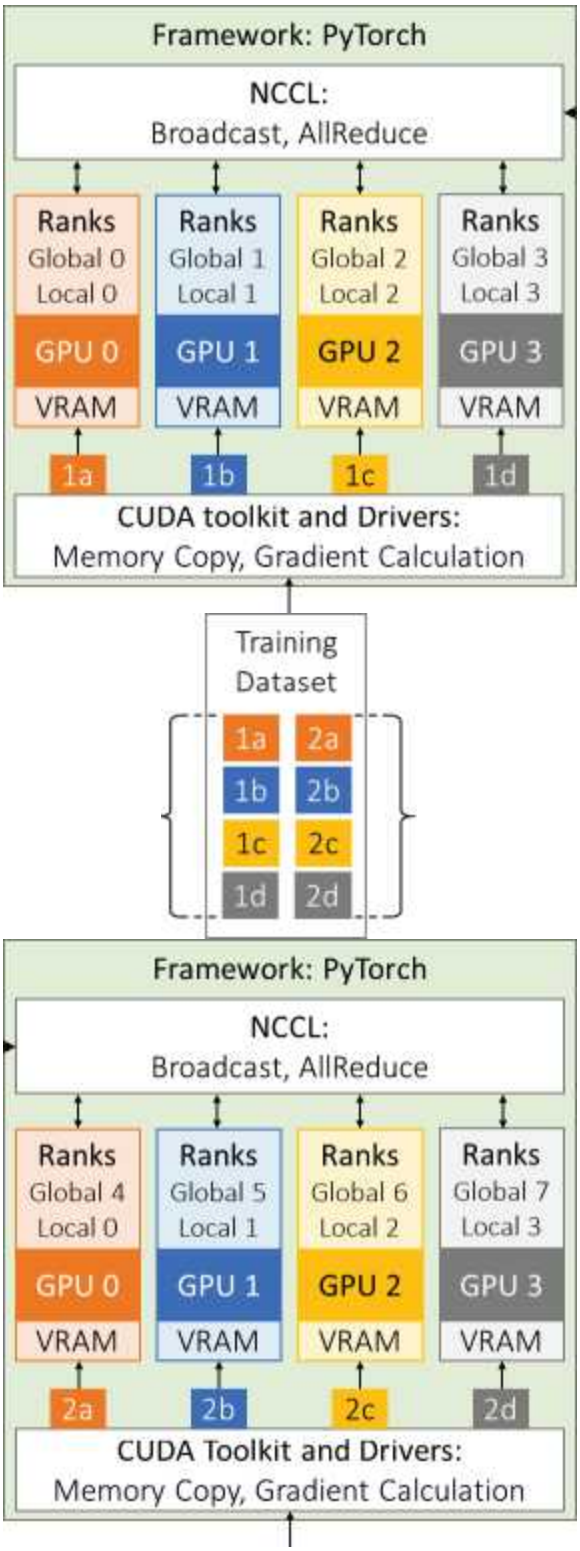


Figure 14-1: *GPU Cluster overall Communication Model.*

DISTRIBUTING NCCL UNIQUE ID FOR GPUS IN A TRAINING CLUSTER

Before GPUs across multiple nodes can communicate efficiently during distributed training, they must first agree on a shared context, a communicator, that defines who is participating and how data will be exchanged. To enable this, the NVIDIA Collective Communications Library (NCCL) requires a special identifier known as the NCCL Unique ID. This ID is generated once by a designated master process and then shared with all other processes in the training job. It acts as a session identifier, ensuring that every participating GPU process joins the same communication group. Without it, there would be no common reference point for building the communication topologies (such as rings or trees) used in operations like AllReduce or Broadcast. In essence, the NCCL Unique ID enables coordinated initialization and makes collective communication possible in a distributed GPU environment. The NCCL Unique ID is distributed over TCP connection. The following two section describes the process.

Opening TCP Socket to Master Node

When the training job is launched using the torchrun command shown in Figure 14-2, PyTorch's distributed framework starts

one process per GPU on each node. These processes are later identified by their global rank ID. Typically, the process with rank ID 0 is assigned the master role, which means it creates and distributes the NCCL Unique ID to all other processes (i.e., ranks running on other GPUs).

The global rank IDs are calculated by multiplying the --node_rank=*n* variable with the --nproc_per_node=4 value (processes per node) and then adding the local GPU rank. This results in the following global rank IDs:

Host A: GPU 0 - Rank ID: 0 * 4 + 0 = 0

Host A: GPU 1 - Rank ID: 0 * 4 + 1 = 1

Host A: GPU 2 - Rank ID: 0 * 4 + 2 = 2

Host A: GPU 3 - Rank ID: 0 * 4 + 3 = 3

Host B: GPU 0 - Rank ID: 1 * 4 + 0 = 4

Host B: GPU 1 - Rank ID: 1 * 4 + 1 = 5

Host B: GPU 2 - Rank ID: 1 * 4 + 2 = 6

Host B: GPU 3 - Rank ID: 1 * 4 + 3 = 7

Since GPU 0 on Host A (node rank 0) has the global rank ID 0, it becomes the master rank. PyTorch opens a TCP listener on this GPU at 192.168.10.101:12345, using the values of --master_addr=192.168.10.101 and --master_port=12345. The script parameter --nnodes=2 specifies that there are two nodes in the cluster, and

--nproc_per_node=4 indicates that four processes (one per GPU) are running on each node. Armed with this information, the master rank expects 7 connection requests (from ranks 1 through 7).

All other ranks start a three-way handshake process for opening TCP socket with the master process. The ranks 4-7 on the host B use --master_addr = 192.168.10.101 as a destination IP address, while local ranks 1-3 use the loopback IP address 127.0.0.1. All ranks use the --master_port=12345 as destination TCP port.

This connection phase enables a rendezvous process, during which the master distributes the NCCL Unique ID so that GPUs can form a communication topology for collective operations.

There is a loose but useful analogy between the NCCL rendezvous process and the Rendezvous Point (RP) in Layer 3 multicast networking. In both cases, the rendezvous acts as a coordination mechanism. The NCCL rendezvous process distributes a unique NCCL identifier from a master process to all GPU processes participating in distributed training. Similarly, a multicast RP serves as a shared point which distributes the data frames from the sender to multiple receivers.

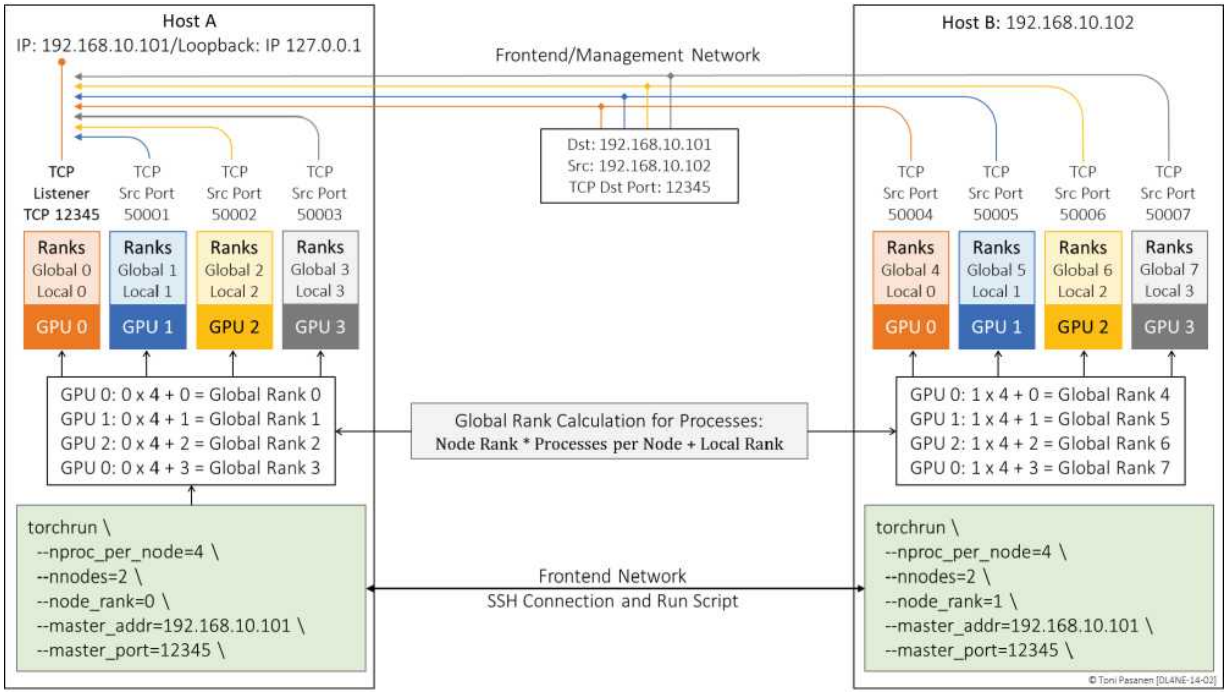


Figure 14-2: *Opening TCP Socket with the Master Rank.*

Distributing the NCCL Unique ID Over Established TCP Sockets

Once all ranks have established TCP connections with the master process (rank ID 0), the next step is to distribute the NCCL Unique ID. The master process generates this identifier and sends it to all other ranks (1 through 7) using the already established TCP connections.

These connections typically run over the frontend or management network, depending on the cluster configuration. Figure 14-3 illustrates the state of the connections from the perspective of the master process.

The NCCL Unique ID serves as a namespace identifier for the job, ensuring that only processes belonging to the same training

job can participate in the same communication group. This is particularly important in environments where multiple distributed jobs may be running concurrently on the same set of nodes. The unique ID isolates jobs from each other and prevents cross-talk between unrelated processes. All ranks then use this unique ID to initialize their local NCCL communicators, ensuring they join the same communication group. This distribution phase is critical: the NCCL Unique ID acts as a bootstrap mechanism, allowing all GPU processes to join the same communication group and participate in collective operations like all-reduce and broadcast.

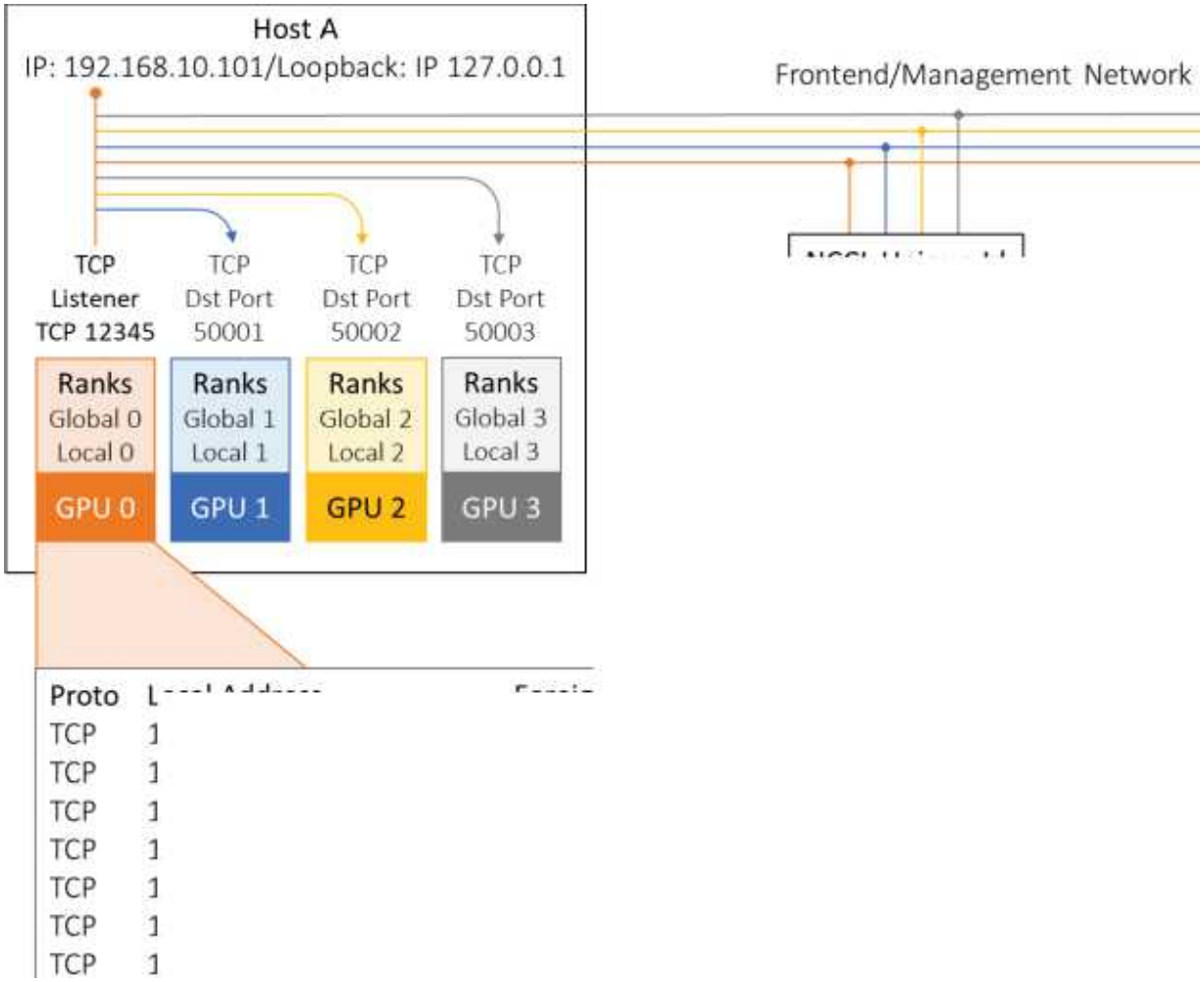
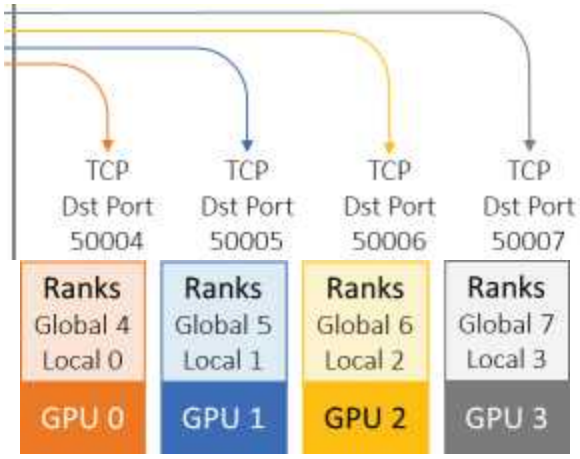


Figure 14-3: *Redistribution NCCL unique Id over TCP Socket.*



NCCL Broadcast Collective and Model Parameter Synchronization

At this point, each process already has a local copy of the model, and all GPUs are ready to begin synchronized training. The first step is to ensure that every GPU starts with identical model parameters. NCCL handles this automatically, using the chosen communication topology.

After the master process (rank 0, running on GPU 0 of Host A) shares the NCCL Unique ID with all other processes over TCP sockets, the NCCL library builds a tree topology. This topology is used for sending model parameters to all other GPUs using the Broadcast collective. Figure 14-4 illustrates how the master process, running on GPU 0 of Host A, distributes its model parameters to all other processes. GPUs with global rank IDs 1-3 are on the same host as the master process, so NCCL uses direct memory copy over high-speed NVLink. These transfers happen without involving the CPU or operating system, and no Queue

Pairs are needed, making intra-node communication extremely fast and efficient.

If GPUs are located on different hosts, NCCL sets up Queue Pairs (QPs) to create fast, direct data paths between the master process and remote processes. These connections use the backend network, which in our example is a routed Layer 3 Clos Fabric (the network layout is excluded for simplicity).

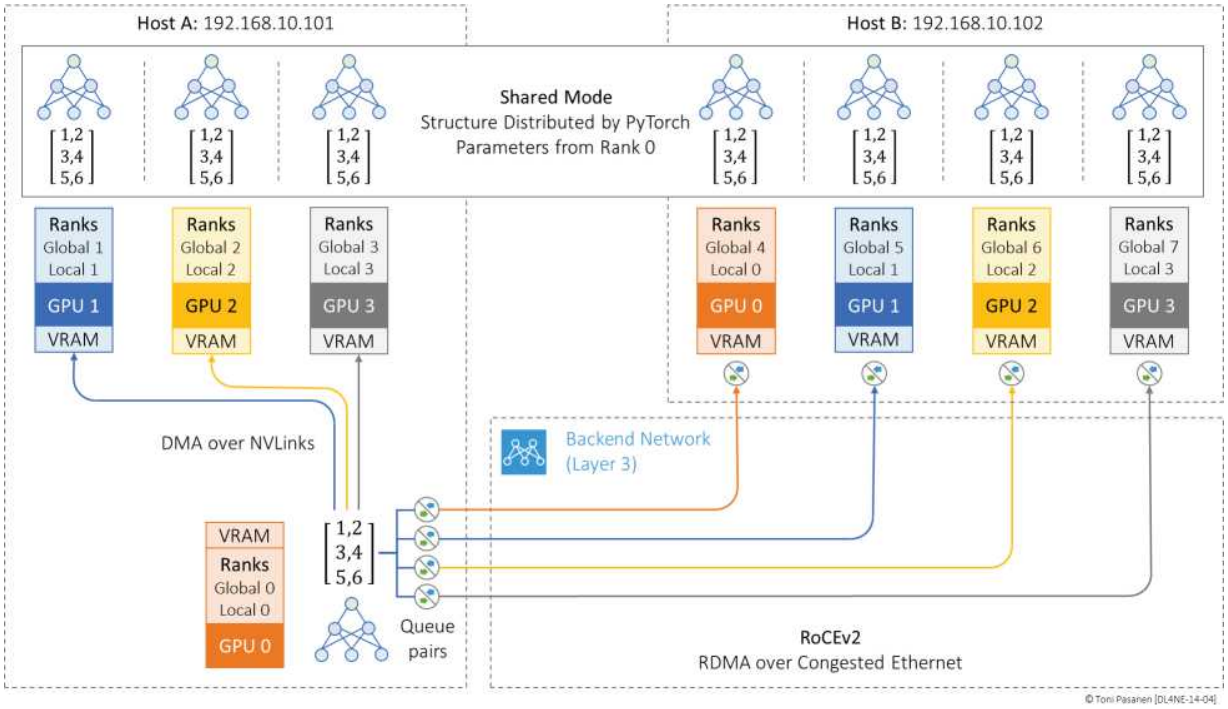


Figure 14-4: *Model Parameters Distribution by Master Rank 0.*

Gradient Synchronization Using AllReduce Collective

After synchronizing model parameters, the first iteration of the forward pass begins simultaneously on all GPUs in the cluster. During the forward pass, GPU-specific mini-batches are processed through all layers of the model by performing matrix

multiplications followed by activation function operations at each layer. After computing the model output y, each GPU starts the backward pass. Let's assume the last layer has 1024 parameters and each GPU computes gradients for all 1024 parameters. These gradients are stored in a reserved memory region called a bucket.

Next, each GPU divides its bucket into four chunks, each containing 256 gradients (since 1024 parameters / 4 GPUs = 256 gradients per chunk). At this point, every GPU has four chunks labeled A–D, each with 256 gradients. When using the AllReduce collective in a unidirectional ring topology, the operation is implemented as ReduceScatter followed by AllGather.

In our example, shown in figure 14-5, we have two nodes (Host A and Host B), each with four GPUs. Every GPU has computed all 1024 gradients and organized them into four local chunks (A–D) in VRAM. In this example, GPU 0 (with global rank 0, we'll use global ranks from now on) is responsible for averaging chunk A, rank 1 (Blue GPU 1) for chunk B, rank 2 (Green GPU 0) for chunk C, and Rank 3 (Yellow GPU 1) for chunk D. Intra-node GPU connections use high-speed NVLink, while inter-node connections use RoCEv2.

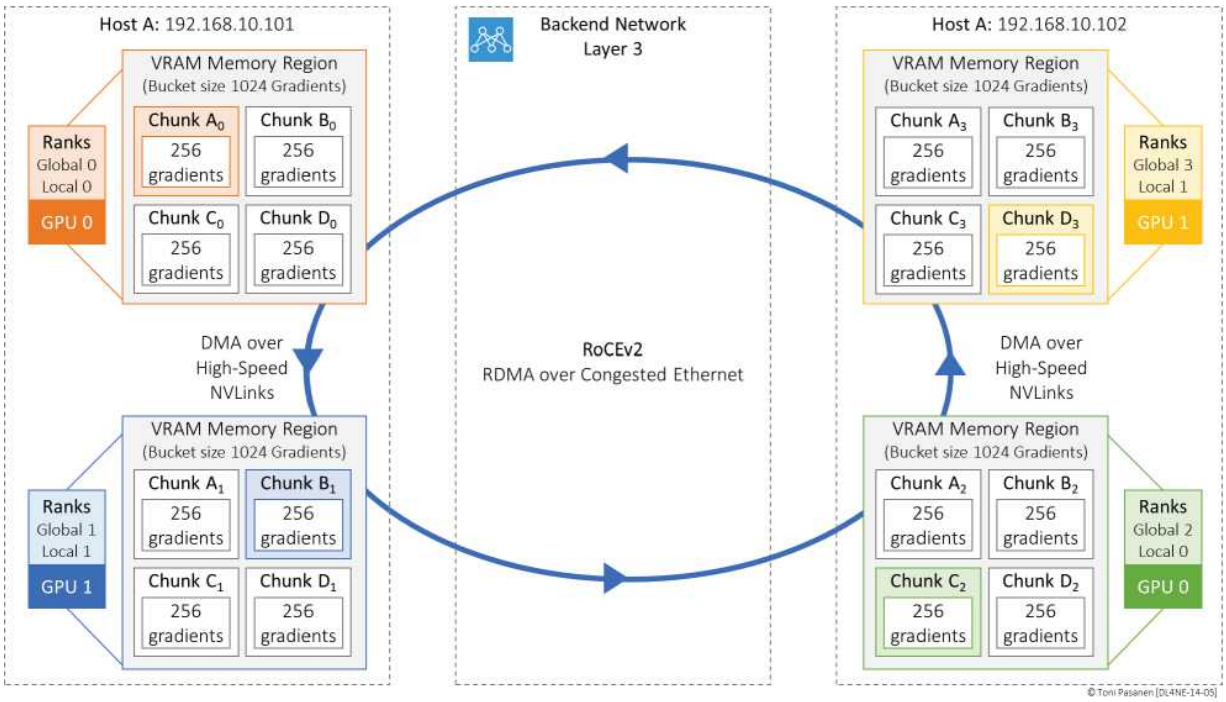


Figure 14-5: *AllReduce in Ring Topology with ReduceScatter and AllReduce Operations*.

ReduceScatter: First Iteration

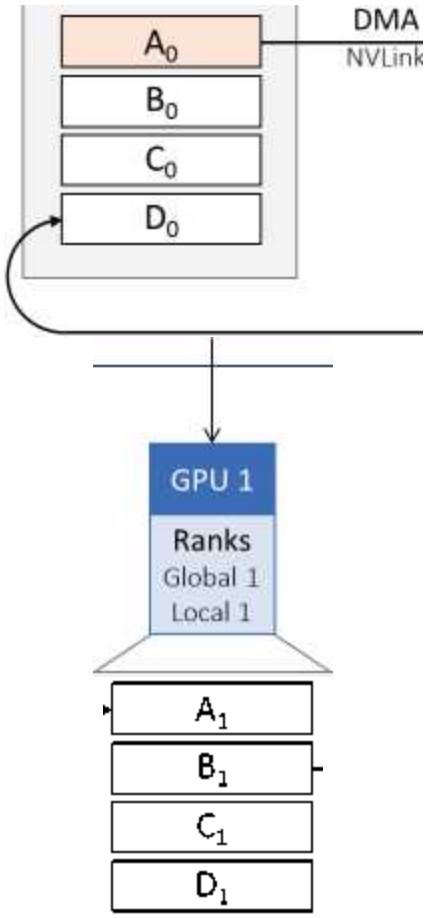
In Figure 14-6, the ring topology from Figure 14-5 is still in use, but the GPUs are laid out in a linear sequence for easier visualization of the AllReduce data flow.

During the ReduceScatter phase of the AllReduce operation, each rank sends the chunk it is responsible for to the next rank in the ring:

- Rank 0 sends chunk A0 to Rank 1
- Rank 1 sends chunk B1 to Rank 2
- Rank 2 sends chunk C2 to Rank 3

- Rank 3 sends chunk D3 to Rank 0

Each of these chunks contains gradients for a specific portion of the parameter space, and each rank is responsible for reducing (i.e., summing) that portion across all GPUs as data circulates around the ring.



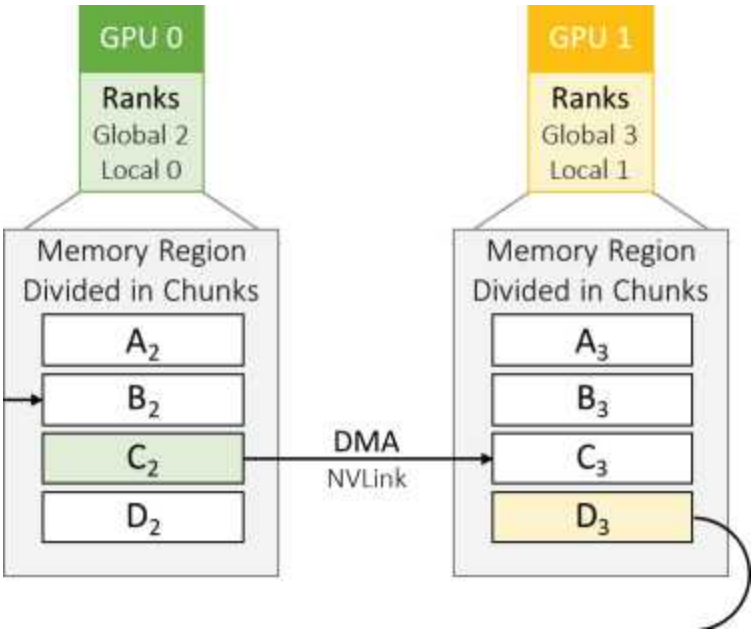


Figure 14-6: *ReduceScatter: The First Iteration.*

Figure 14-7 shows the status of gradient synchronization after the first send iteration in the ReduceScatter phase. At this point, each GPU has sent its assigned chunk to the next GPU in the ring topology and has also received one chunk from its neighbor:

- Rank 0 has received chunk D3 from rank 3
- Rank 1 has received chunk A0 from rank 0
- Rank 2 has received chunk B1 from rank 1
- Rank 3 has received chunk C2 from rank 2

After this first send (iteration 1 of the ReduceScatter phase), each GPU still holds three original chunks in local memory, plus one partially reduced chunk. Each GPU adds the received chunk to its local version of the same chunk. For example, rank 0 adds chunk D3 to D0 (chunk D = D0 + D3).

This is only the first partial reduction. To complete the full reduction for its assigned chunk, each GPU must receive and sum the corresponding chunks from the remaining GPUs over the next three iterations. By the end of the ReduceScatter phase (after three iterations in a 4-GPU ring), each GPU holds exactly one fully reduced chunk, though not necessarily the one it originally owned.

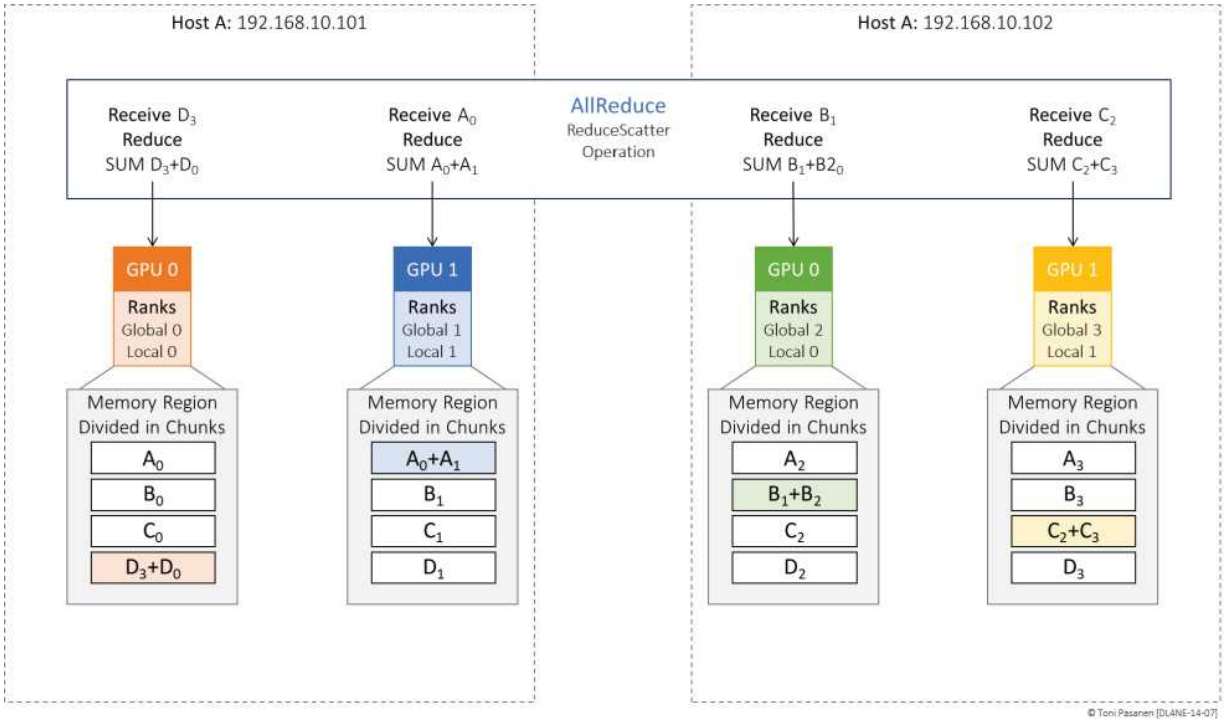


Figure 14-7: *ReduceScatter: Chunks After the First Iteration.*

ReduceScatter: Second Iteration

The figure 14-8 shows the second iteration of the ReduceScatter phase:

- Rank 0 sends the partially averaged chunk D (Sum of D3 + D0) to Rank 1.

- Rank 1 sends the partially averaged chunk A (Sum of A0 + A1) to Rank 2.
- Rank 2 sends the partially averaged chunk B (Sum of B1 + B2) to Rank 3.
- Rank 3 sends the partially averaged chunk C (Sum of C2 + C3) to Rank 0.

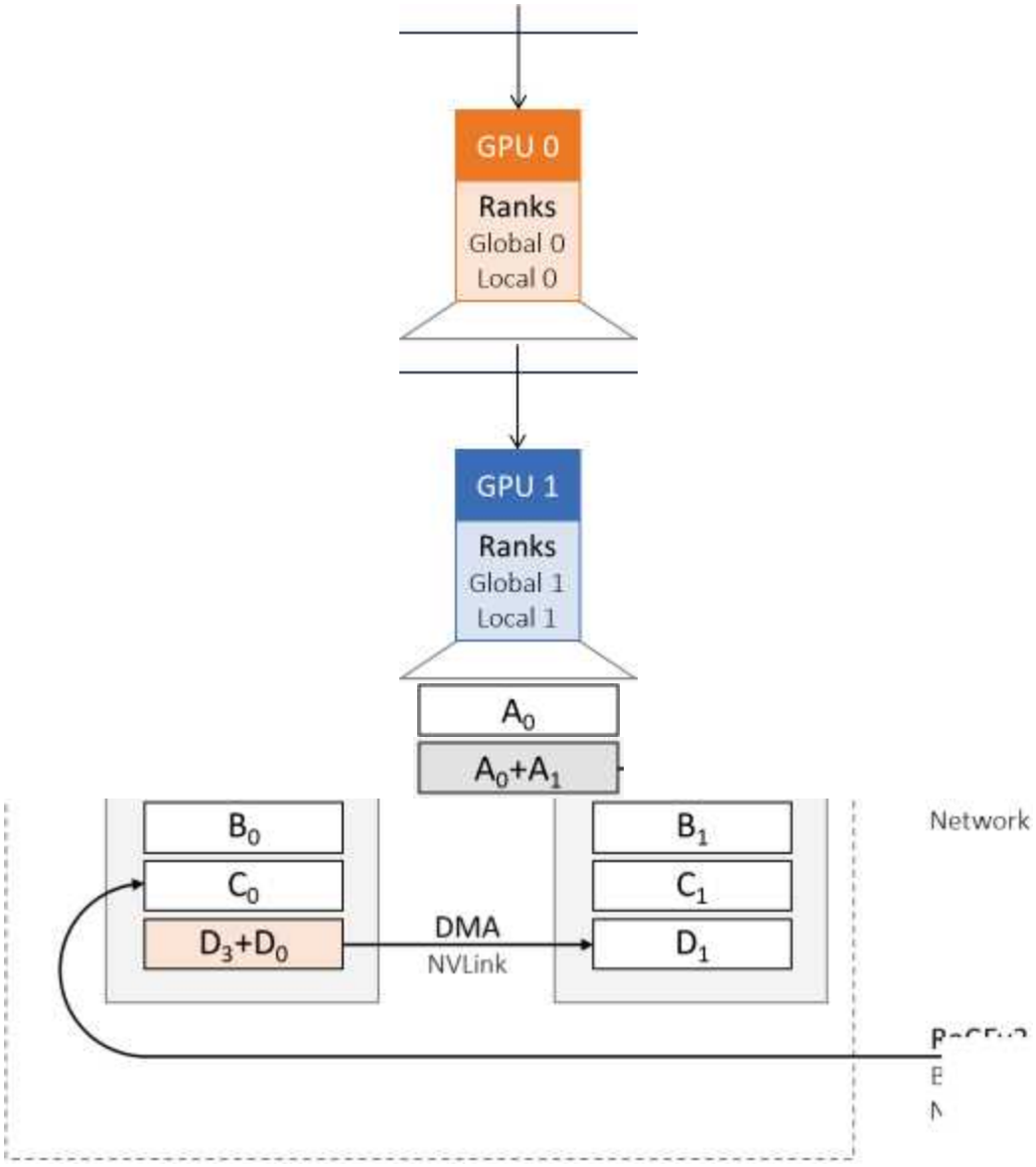
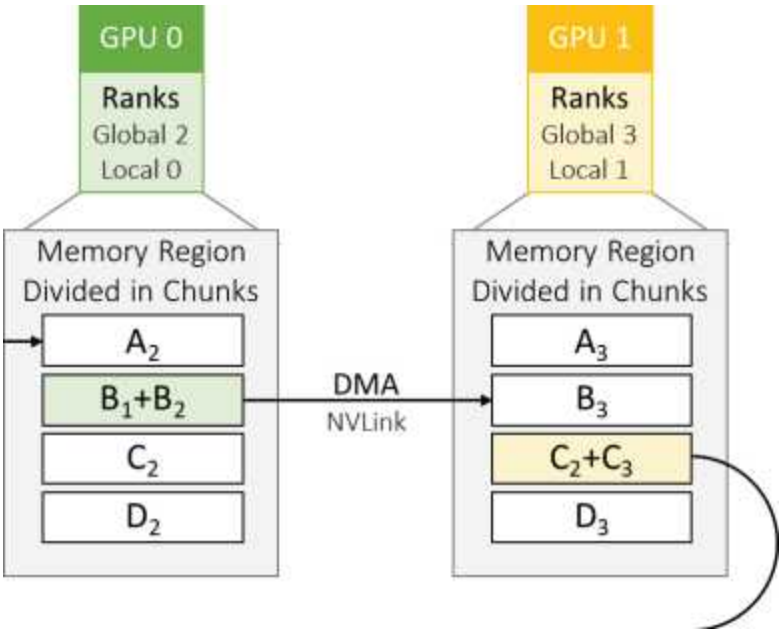


Figure 14-8: *ReduceScatter: The Second Iteration.*



After the second ReduceScatter iteration, each rank now holds:

- One chunk that has been partially reduced twice (local chunk + two remote chunks)
- Two original chunks that have not yet been involved in any communication
- One chunk that was sent out during this iteration

Here's the specific status per rank:

Rank 0:

- Holds partially reduced chunk C = C2 + C3 + C0 (just received from Rank 3 and added to local C0)
- Still has original chunks A0 and B0
- Sent out chunk D = D3 + D0

Rank 1:

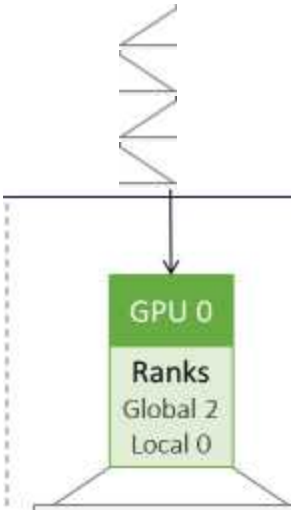
- Holds partially reduced chunk D = D3 + D0 + D1
- Still has original chunks B1 and C1
- Sent out chunk A = A0 + A1

Rank 2:

- Holds partially reduced chunk A = A0 + A1 + A2
- Still has original chunks C2 and D2
- Sent out chunk B = B1 + B2

Rank 3:

- Holds partially reduced chunk B = B1 + B2 + B3
- Still has original chunks A3 and D3
- Sent out chunk C = C2 + C3



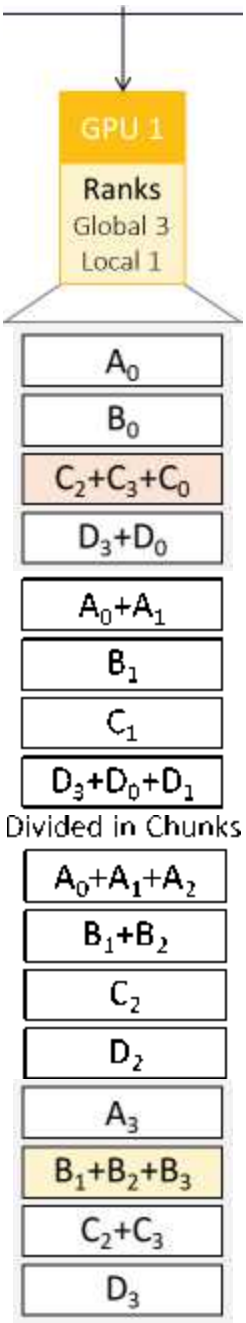
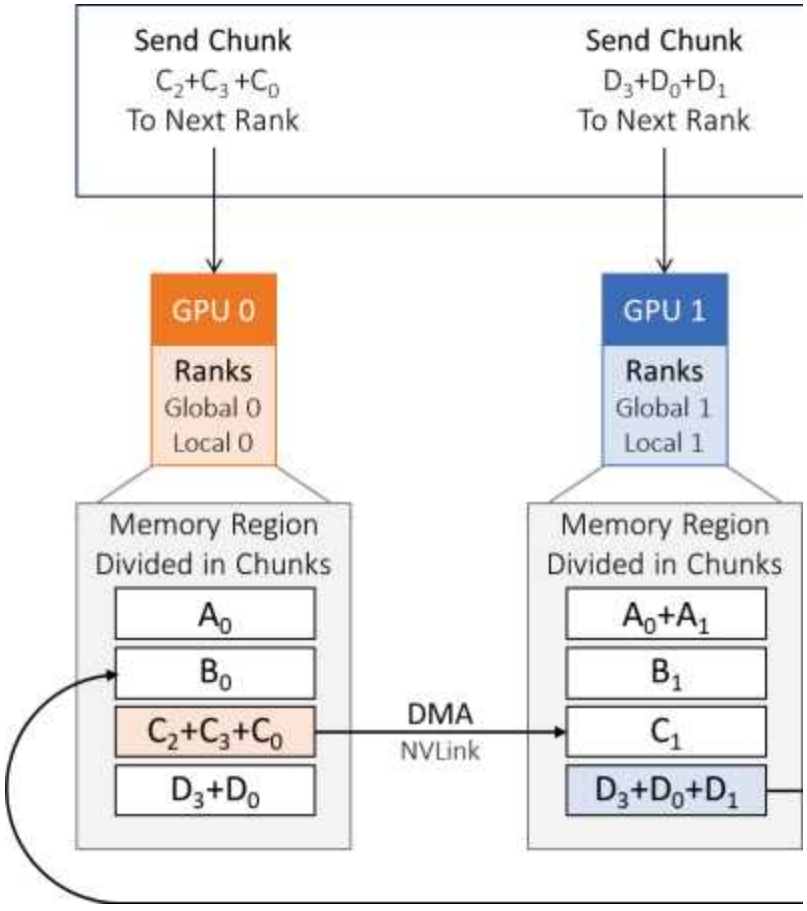


Figure 14-9: *ReduceScatter: Chunks After the Second Iteration.*

ReduceScatter: Third Iteration

The figure 14-10 shows the third iteration of the ReduceScatter phase:

- Rank 0 sends the partially averaged chunk C (Sum of C2 + C3+ C0) to Rank 1.
- Rank 1 sends the partially averaged chunk D (Sum of D3 + D0 + D1) to Rank 2.
- Rank 2 sends the partially averaged chunk A (Sum of A0 + A1 + A2) to Rank 3.
- Rank 3 sends the partially averaged chunk B (Sum of B1 + B2 + B3) to Rank 0.



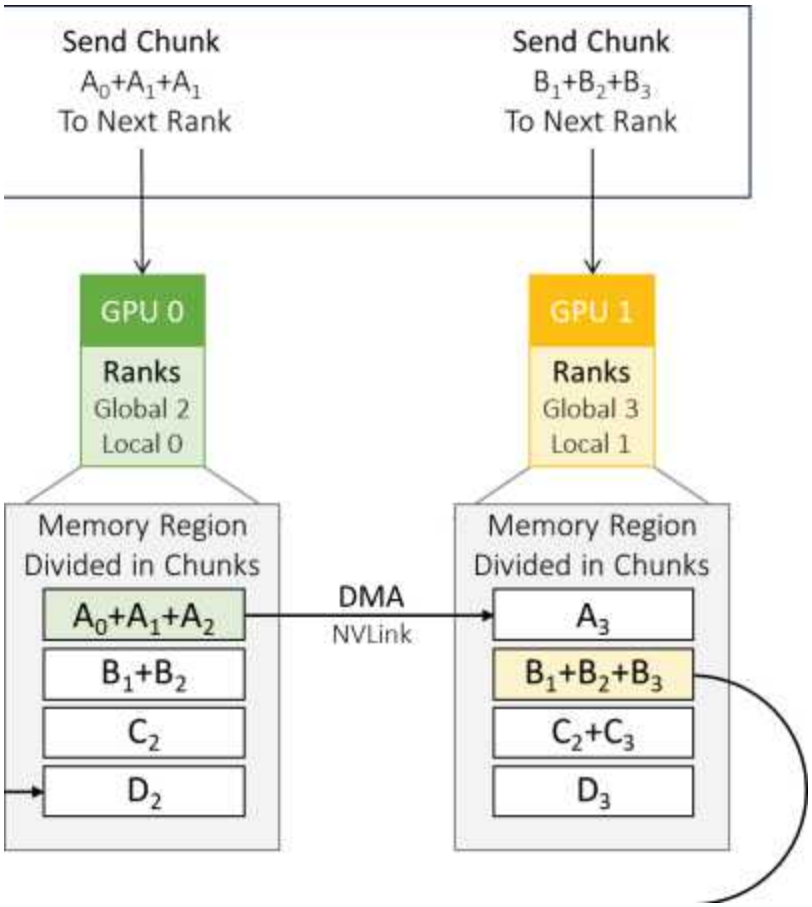


Figure 14-10: *ReduceScatter: the Third Iteration.*

After the third ReduceScatter iteration, the ReduceScatter phase is complete. Each rank now holds one fully reduced chunk, which includes contributions from all four GPUs. However, these fully reduced chunks are not located on their original owner ranks. Each rank receives one final chunk and completes its reduction by summing it with its local copy. At this point:

- Rank 0 holds fully reduced chunk B = B1 + B2 + B3 + B0
- Rank 1 holds fully reduced chunk C = C2 + C3 + C0 + C1
- Rank 2 holds fully reduced chunk D = D3 + D0 + D1 + D2

- Rank 3 holds fully reduced chunk A = A0 + A1 + A2 + A3

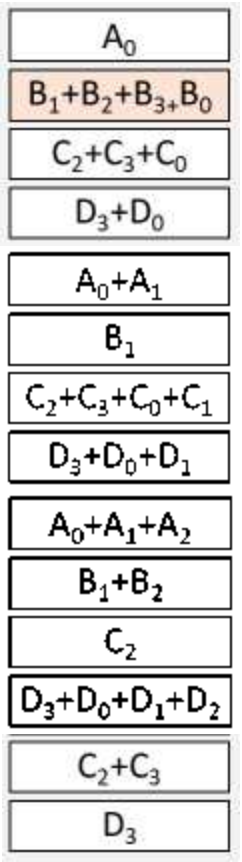


Figure 14-11: *ReduceScatter: Chunks After the Third Iteration.*

The ReduceScatter operation is the first step in the Ring AllReduce process and is responsible for aggregating gradient data across all GPUs in a distributed training setup. Its goal is to compute the sum (or average) of each gradient. At the end of the ReduceScatter phase:

- Each GPU holds one fully reduced chunk, which includes contributions from all four GPUs.
- The reduced chunk is not necessarily local to the GPU that now holds it, it's owned by a different rank.

AllGather: The first Iteration

Now that the ReduceScatter phase has completed, the AllGather phase begins. Its job is to distribute the fully reduced chunks back to all GPUs, so that each one ends up with a complete, synchronized copy of all gradients, ready to be used to update the model.

In the first iteration of AllGather:

- Each GPU sends the fully reduced chunk it currently holds to the next GPU in the ring.
- At the same time, it receives a new reduced chunk from the previous GPU.

Here's what happens on each rank during this iteration:

- Rank 0 sends reduced chunk B to Rank 1 and receives chunk A from Rank 3.
- Rank 1 sends reduced chunk C to Rank 2 and receives chunk B from Rank 0.
- Rank 2 sends reduced chunk D to Rank 3 and receives chunk C from Rank 1.
- Rank 3 sends reduced chunk A to Rank 0 and receives chunk D from Rank 2.

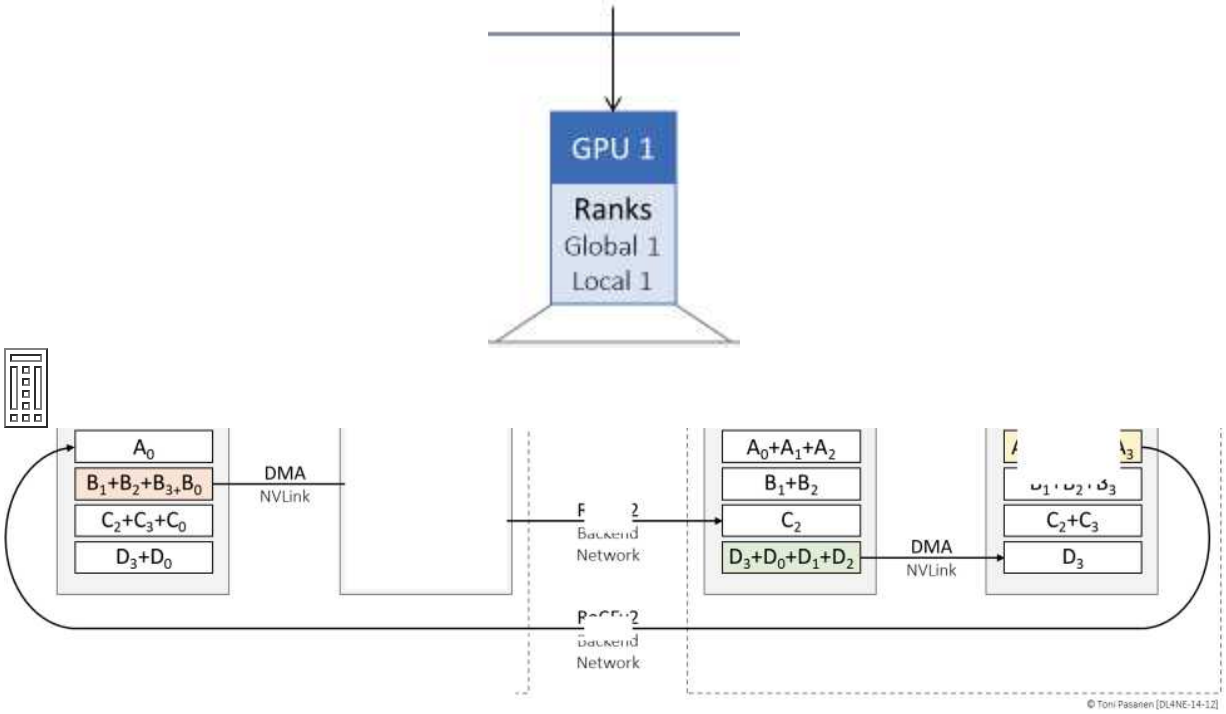


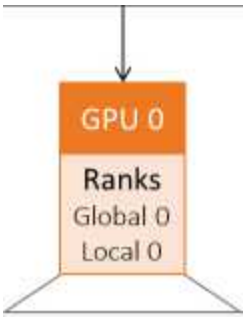
Figure 14-12: *AllGather: the First Iteration.*

Each GPU now holds two reduced chunks: the one it originally reduced

during ReduceScatter, and one received from its neighbor. This process

continues for three iterations, after which all GPUs will have a complete,

fully averaged set of gradients.



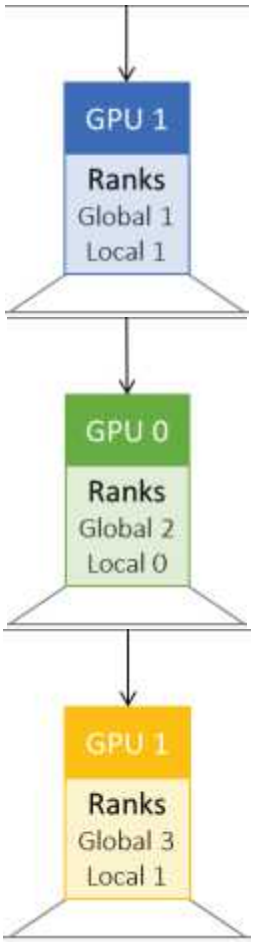


Figure 14-13: *AllGather: Chunks After the First Iteration.*

AllGather: The Second Iteration

In the second iteration, each GPU again sends the most recently received chunk to the next GPU in the ring. This continues the process of distributing fully reduced gradient chunks to all peers.

Here's what happens:

- Rank 0 sends chunk A (received from Rank 3 in iteration 1) to Rank 1

- Rank 1 sends chunk B (received from Rank 0) to Rank 2
- Rank 2 sends chunk C (received from Rank 1) to Rank 3
- Rank 3 sends chunk D (received from Rank 2) to Rank 0

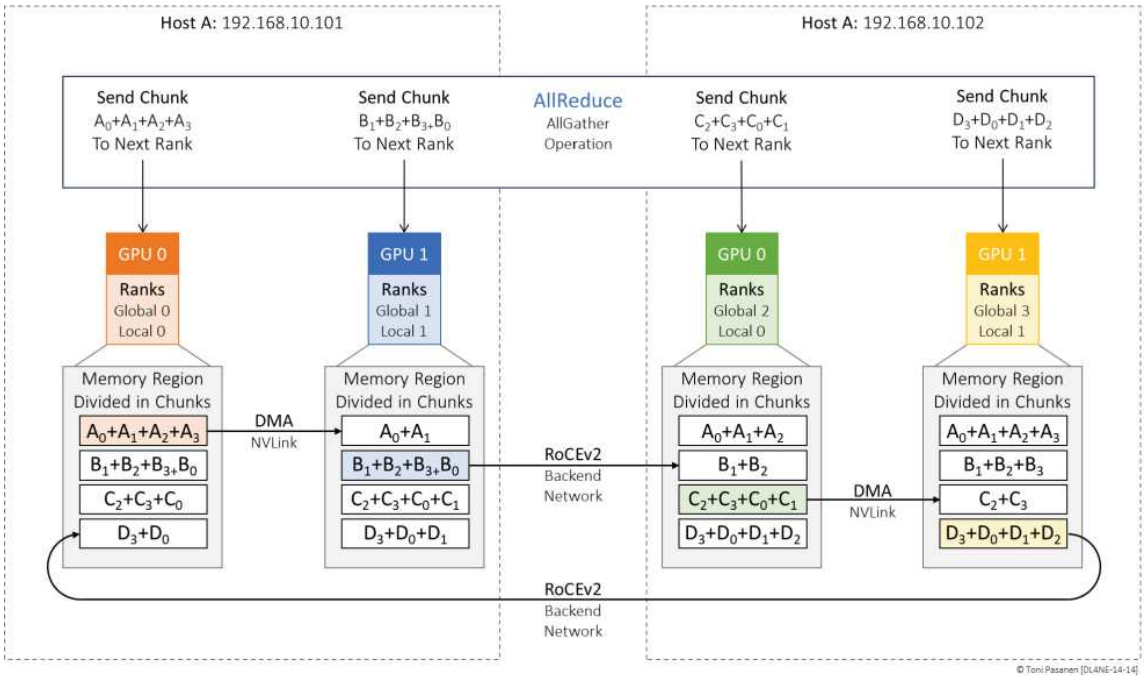
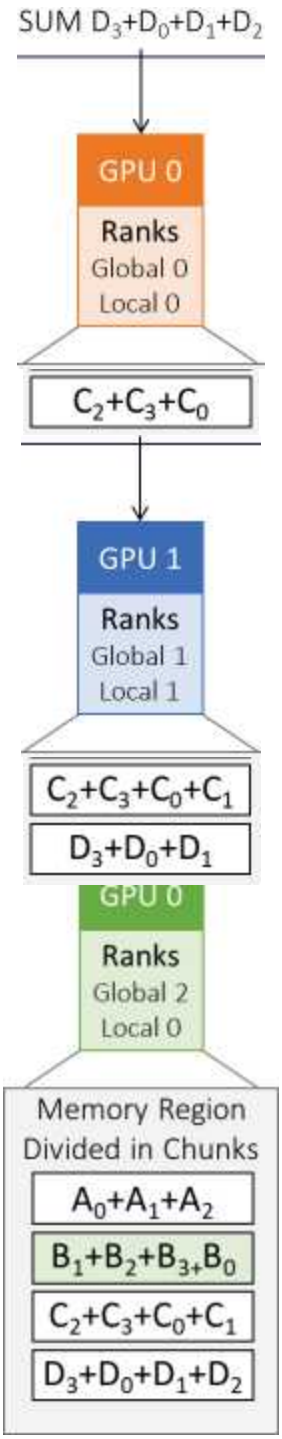


Figure 14-14: *AllGather: the Second Iteration.*

Each GPU now holds three fully reduced chunks:

- Rank 0 has: A (original), B, and D
- Rank 1 has: B (original), C, and A
- Rank 2 has: C (original), D, and B
- Rank 3 has: D (original), A, and C

Only one chunk is still missing per GPU, which will be received in the third and final AllGather iteration, completing the synchronization.


SUM $D_3+D_0+D_1+D_2$
GPU 0
Ranks
Global 0
Local 0
$C_2+C_3+C_0$
GPU 1
Ranks
Global 1
Local 1
$C_2+C_3+C_0+C_1$
$D_3+D_0+D_1$
GPU 0
Ranks
Global 2
Local 0
Memory Region
Divided in Chunks
$A_0+A_1+A_2$
$B_1+B_2+B_{3+}B_0$
$C_2+C_3+C_0+C_1$
$D_3+D_0+D_1+D_2$

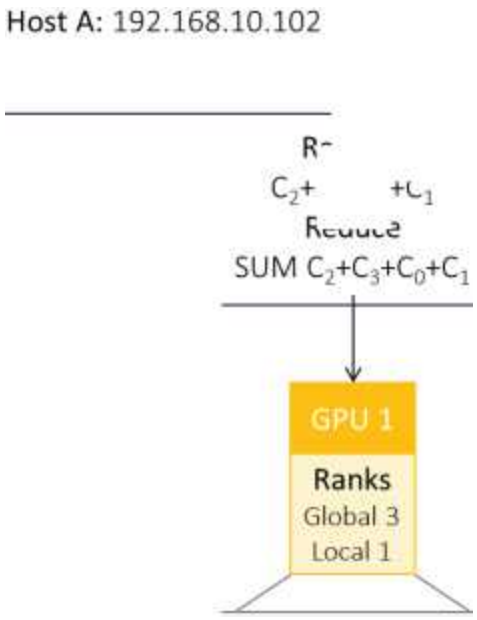


Figure 14-15: *AllGather: Chunks After the Second Iteration.*

AllGather: Third Iteration

In the final AllGather iteration, each GPU sends the chunk it received during the second iteration to the next GPU in the ring. After this operation, all GPUs have a complete set of synchronized gradient chunks.

Here's what happens:

- Rank 0 sends chunk D to Rank 1
- Rank 1 sends chunk A to Rank 2
- Rank 2 sends chunk B to Rank 3
- Rank 3 sends chunk C to Rank 0

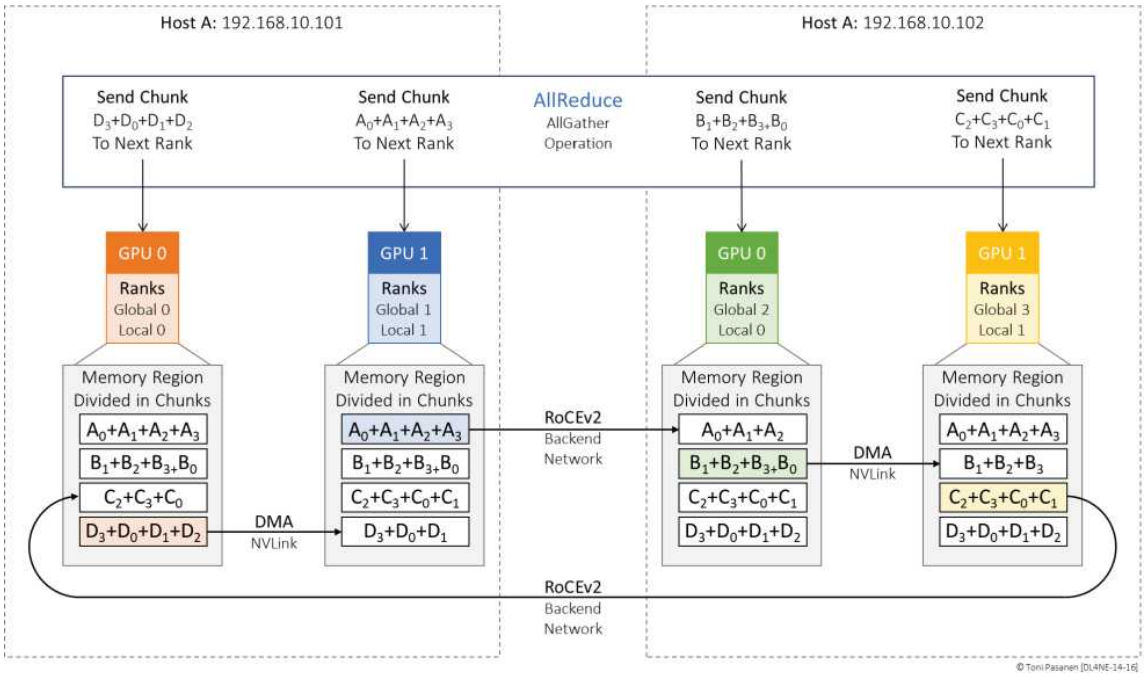


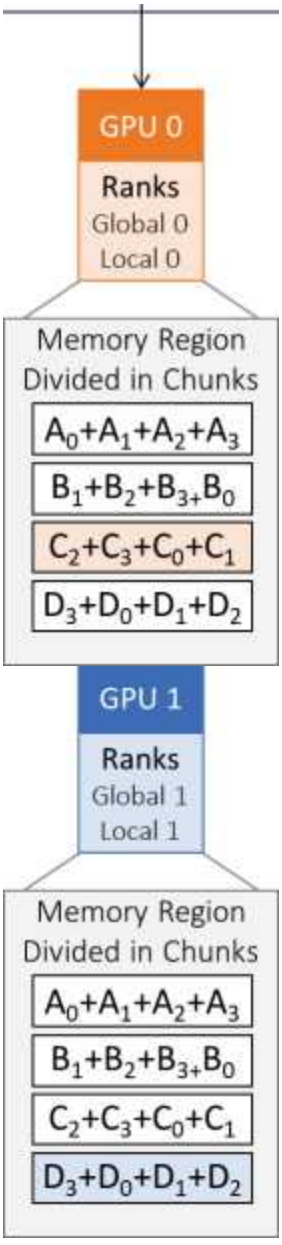
Figure 14-16: *AllGather: the Third Iteration.*

At this point, each GPU has received all four reduced chunks (A, B, C, and D), and all GPUs now have a complete set of 1024 gradients, each fully reduced (summed across all GPUs).

The AllReduce process computes the sum of each gradient across all GPUs, but in data-parallel training, we usually want the average gradient. So, after receiving all four chunks (each containing 256 gradients), every GPU performs element-wise division by the number of GPUs (which is 4 in your setup). This means:

- Each of the 1024 gradients is divided by 4.
- The result is the average gradient, which represents the combined learning signal from all GPUs' local mini-batches.

These averaged gradients are then used to update the model weights locally, and since all GPUs now have the same gradient values, each model replica remains perfectly synchronized.



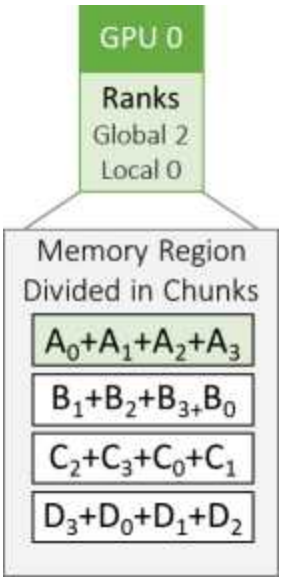


Figure 14-17: *AllGather: All Chunks Synchronized and Averaged.*

Finalizing the AllReduce Operation

At the end of the AllGather phase, each GPU holds a complete, fully reduced set of all gradients, in our example, 1024 values that represent the sum of corresponding gradients from all GPUs. This means the synchronization is now complete: all GPUs have identical gradient vectors, and model consistency across the cluster is guaranteed. However, the goal of distributed training is typically to compute the average gradient, not the sum. To achieve this, each GPU simply divides each of the 1024 gradient values by the number of participating GPUs, in our case, four. This is a local operation, performed independently on each GPU, without further communication.

Because all GPUs perform this averaging on the same synchronized data, the result remains consistent across the cluster. No additional synchronization is needed after this step.

The model is now ready for a consistent weight update across all GPUs, and the next training iteration can begin.

REFERENCES

[1] Karthik Mandakolathur, Sylvain Jeaugey, “Doubling all2all Performance with NVIDIA Collective Communication Library 2.12”, February 28, 2022. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/

[2] Sylvain Jeaugey, Giuseppe Congiu, Thomas Gillis, Ben Williams, Fred Oh, “Fast Multi-GPU collectives with NCCL”, 2018.

https://developer.nvidia.com/blog/fast-multi-gpu-collectives-nccl

[3] Sylvain Jeaugey, Giuseppe Congiu, Thomas Gillis, Ben Williams, Fred Oh, “New Scaling Algorithm and Initialization with NVIDIA Collective Communications Library 2.23”, January 31, 2025.

https://developer.nvidia.com/blog/new-scaling-algorithm-and-initialization-with-nvidia-collective-communications-library-2-23/

[4] Ben Williams, Kaiming Ouyang, Kamil Iskra, Sylvain Jeaugey, “Networking Reliability and Observability at Scale

with NCCL 2.24", March 13, 2025. https://developer.nvidia.com/blog/networking-reliability-and-observability-at-scale-with-nccl-2-24/

[5] Andreas Jocksch, Noe Ohana, Emmanuel Lanti, Vasileios Karakasis, Laurent Villard, "Optimised allgatherv, reduce_scatter and allreduce communication in message-passing systems", June 23, 2020.

https://arxiv.org/abs/2006.13112

[6] Jesper Larsson Träff, "Optimal, Non-pipelined Reduce-scatter and Allreduce

Algorithms", October 18, 2024.

https://arxiv.org/abs/2410.14234

[7] Thomas B. Rolinger, Tyler A. Simon, Christopher D. Krieger, "An Empirical Evaluation of Allgatherv on Multi-GPU Systems", December 14, 2018. https://arxiv.org/abs/1812.05964

[8] Dian Xiong, Li Chen, Youhe Jiang, Dan Li, Shuai Wang, Songtao Wang, "Revisiting the Time Cost Model of AllReduce", September 6, 2024. https://arxiv.org/abs/2409.04202

[9] PyTorch Team, "Start Locally | Getting Started with PyTorch", May 9, 2025. https://pytorch.org/get-started/locally/

[10] PyTorch Team, “torch.distributed.all_reduce — PyTorch Distributed

Package”, May 9, 2025.

https://pytorch.org/docs/stable/distributed.html#torch.distributed.all_reduce

[11] NVIDIA Corporation, “CUDA Toolkit Downloads”, May 9, 2025.

https://developer.nvidia.com/cuda-downloads

[12] NVIDIA Corporation, “CUDA C++ Programming Guide”, May 9, 2025.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[13] NVIDIA Corporation, “NCCL Installation Guide”, May 9, 2025.

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/installation.html

[14] NVIDIA Corporation, “NCCL GitHub Repository”, May 9, 2025.

https://github.com/NVIDIA/nccl

BACK COVER TEXT

Deep Learning for Network Engineers bridges the gap between AI theory and modern data center network infrastructure. This book offers a technical foundation for network professionals who want to understand how Deep Neural Networks (DNNs) operate—and how GPU clusters communicate at scale.

Part I (Chapters 1–8) explains the mathematical and architectural principles of deep learning. It begins with the building blocks of artificial neurons and activation functions, and then introduces Feedforward Neural Networks (FNNs) for basic pattern recognition, Convolutional Neural Networks (CNNs) for more advanced image recognition, Recurrent Neural Networks (RNNs) for sequential and time-series prediction, and Transformers for large-scale language modeling using self-attention. The final chapters present parallel training strategies used when models or datasets no longer fit into a single GPU. In data parallelism, the training dataset is divided across GPUs, each processing different mini-batches using identical model replicas. Pipeline parallelism segments the model into

sequential stages distributed across GPUs. Tensor (or model) parallelism further divides large model layers across GPUs when a single layer no longer fits into memory.These approaches enable training jobs to scale efficiently across large GPU clusters.

Part II (Chapters 9–14) focuses on the networking technologies and fabric designs that support distributed AI workloads in modern data centers. It explains how RoCEv2 enables direct GPU-to-GPU memory transfers over Ethernet, and how congestion control mechanisms like DCQCN, ECN, and PFC ensure lossless high-speed transport. You'll also learn about AI-specific load balancing techniques, including flow-based, flowlet-based, and per-packet spraying, which help avoid bottlenecks and keep GPU throughput high. Later chapters examine GPU collectives such as AllReduce—used to synchronize model parameters across all workers— alongside ReduceScatter and AllGather operations. The book concludes

with a look at rail-optimized topologies that keep multi-rack GPU clusters efficient and resilient.

This book is not a configuration or deployment guide. Instead, it equips you with the theory and technical context needed to begin deeper study or participate in cross-disciplinary conversations with AI engineers and systems designers. Architectural diagrams and practical examples clarify complex processes—without diving into implementation details.

Readers are expected to be familiar with routed Clos fabrics, BGP EVPN control planes, and VXLAN data planes. These

technologies are assumed knowledge and are not covered in the book.

Whether you're designing next-generation GPU clusters or simply trying to understand what happens inside them, this book provides the missing link between AI workloads and network architecture.

Page left blank intentionally

Page left blank intentionally

Page left blank intentionally

Page left blank intentionally