DSC204A: Scaleable Data Systems Spring 2025

An Introduction to ML Sys

Instructor: Umesh Bellur *Scribe: Arya Gaikwad, Shruti Sawant, Nishanth Ramesha*

Recap of Previous Lecture: Dan Talk

The previous lecture by Dan covered several key topics, focusing on the system-level challenges and solutions in modern machine learning.

Flash Attention

- **Premise:** The core idea behind Flash Attention is that many existing systems solutions for large models have not adequately accounted for I/O (Input/Output) bottlenecks.
- **Research Problem:** The specific research problem addressed is that system solutions often fail to consider the latencies involved in memory transfers, particularly between high-bandwidth memory (HBM) on accelerators and main CPU memory.
- **Application:** This method finds its primary application in Transformer models, which are foundational to many state-of-the-art NLP and vision tasks.

Chipmunk Project

Dan also briefly discussed the Chipmunk project, which is related to video generation. The project involves handling large activations that occur during the generative process, specifically in denoising steps.

An Introduction to ML Sys

Machine Learning Systems (MLSys) is an emerging field that combines principles from machine learning, systems, and databases to build scalable, efficient, and robust ML platforms. A holistic MLSys approach considers the interplay between data, computational hardware, ML models, and the end-to-end system architecture.

MLSys Elements

The lecture introduced a layered view of ML Systems, showing how a high-level ML model is progressively lowered through different stages of abstraction to be executed on hardware.

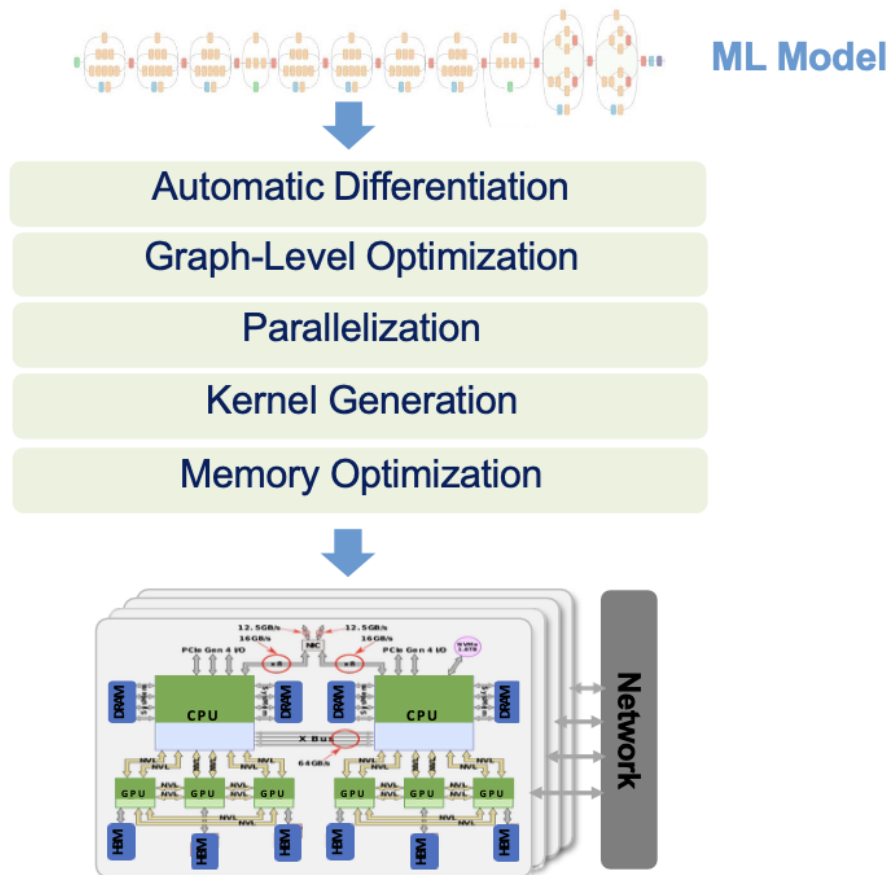

Figure 1: The layers of an ML System, starting from the ML Model and moving down through Automatic Differentiation, Graph-Level Optimization, Parallelization, Code Optimization, and Memory Optimization to the underlying hardware. [Diagram from MLSys-Intro PPT, Slide 2]

Layer 1: Automatic Differentiation

Automatic Differentiation (AD) is a technique to automatically and efficiently compute derivatives of functions represented as computer programs. As noted, you want to minimize a loss function when training a deep learning model, and AD is the tool that makes this feasible for complex models.

- It can compute gradients automatically up to machine precision.
- It is highly optimized for both speed and memory efficiency.

Why is AD relevant in MLSys?

AD is not just a mathematical convenience; it's a cornerstone of modern machine learning infrastructure.

- **Engine of Deep Learning:** AD is the core engine that drives the training of deep learning models via gradient-based optimization.

- **Scalable and Efficient:** It makes the computation of gradients automatic, removing the need for manual derivation, and is highly efficient and scalable.
- **Core System Component:** AD is a fundamental component of all modern ML frameworks (like TensorFlow and PyTorch) and compilers (like XLA and MLIR).
- **Two Modes:** AD operates in two primary modes:
  - **Forward Mode:** Computes derivatives by flowing from inputs to outputs. This is most efficient for functions with a few inputs and many outputs.
  - **Reverse Mode:** Computes gradients by flowing from the final output back to the inputs. This is ideal for functions with many inputs and a single output, which is exactly the case for a typical loss function in ML. Reverse mode AD is also known as **backpropagation**.

Under the Hood of AD

The process of AD involves a few key steps:

1. **Computational Graph:** First, it constructs a computational graph during the forward pass, where nodes represent operations and edges represent the flow of data (tensors).
2. **Chain Rule:** It then applies the chain rule of calculus backward through this graph, starting from the final output (loss), to compute the gradient of the loss with respect to each parameter.
3. **Optimized Execution:** The entire process is optimized for execution on hardware accelerators like GPUs and TPUs.

Single Layer Regression Model Example

Let's walk through a simple example of a single-layer regression model to see AD in action.

The model is defined as:

$$\hat{y} = w \cdot x + b$$

And the loss function is the squared error:

$$L = (\hat{y} - y)^2$$

We use the following initial values:

- Input $x = 2$
- Target $y = 5$
- Weight $w = 1$
- Bias $b = 0$

Forward Pass: First, we calculate the prediction and the loss.

$$\hat{y} = (1 \cdot 2) + 0 = 2$$

$$L = (2 - 5)^2 = (-3)^2 = 9$$

The computation can be broken down into elementary steps, forming a computational graph:

Variable	**Expression**	**Value**
z_1	$w \cdot x$	2
z_2	$z_1 + b$	2
$\hat{y}$	z_2	2
L	$(\hat{y} - y)^2$	9

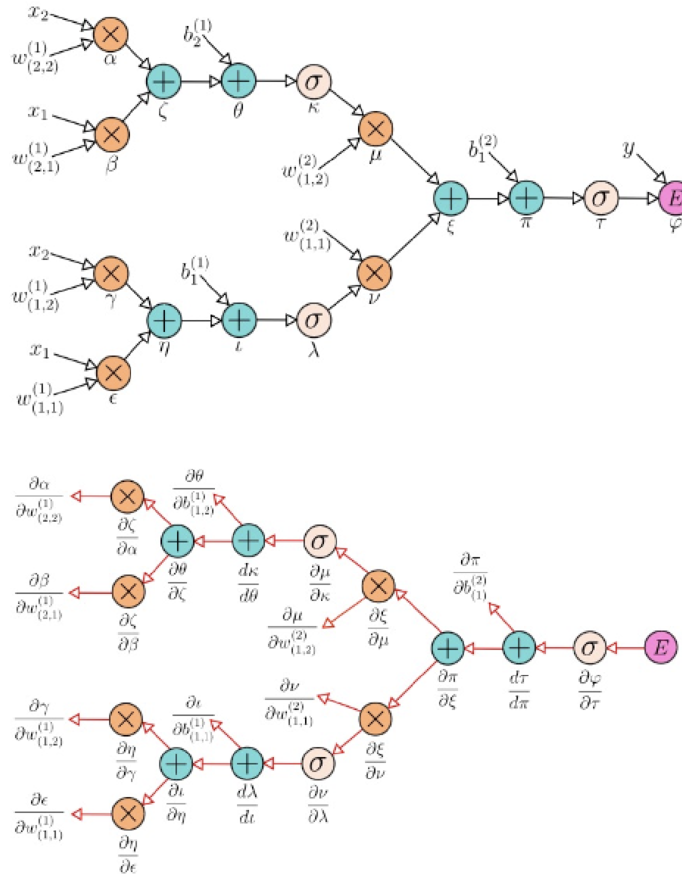

Figure 2: The computational graph for the single-layer regression model. White arrows show the forward pass, and red arrows would represent the backward pass for gradients. [Diagram from MLSys-Intro PPT, Slide 4]

Backward Pass (Applying the Chain Rule): Now, we compute the partial derivatives of each step to find $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$.

Local gradients:

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) = 2(2 - 5) = -6$$

$$\frac{\partial \hat{y}}{\partial z_2} = 1$$

$$\frac{\partial z_2}{\partial z_1} = 1$$

$$\frac{\partial z_2}{\partial b} = 1$$
$$\frac{\partial z_1}{\partial w} = x = 2$$

Using the chain rule, we combine these local gradients to get the gradients of the loss with respect to the parameters w and b:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w}$$
$$= (-6) \cdot 1 \cdot 1 \cdot 2 = -12$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial b}$$
$$= (-6) \cdot 1 \cdot 1 = -6$$

These gradients ($\frac{\partial L}{\partial w} = -12$ and $\frac{\partial L}{\partial b} = -6$) are then used by an optimizer (like SGD) to update the model parameters w and b to minimize the loss.

This example shows how AD chains together simple derivatives to compute complex ones. The entire computational graph must be mapped and traversed. This leads to a natural question for the next layer of the ML Sys stack: **Can we optimize this graph? How can we make it simpler and more efficient to execute?** This will be addressed when discussing graph-level optimizations.

Layer 2: Graph-Level Optimizations

This section covers the fundamental concepts and techniques used in optimizing computational graphs in machine learning systems. Graph-level optimizations sit as a crucial layer between automatic differentiation and parallelization, focusing on transforming computation graphs to achieve better performance before mapping them to hardware.

Introduction to Computational Graphs in ML

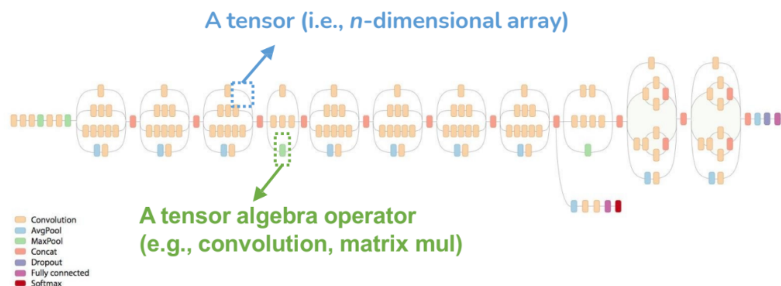

Figure 3: The computational graph with tensor operators. [Diagram from MLSys-Intro PPT, Slide 6]

ML models are fundamentally composed of computational graphs. These graphs consist of tensors (multi-dimensional arrays) and various operations performed on these tensors. The computational flow involves

a series of operators being applied sequentially or in parallel, including operations like convolution, ReLU activations, matrix multiplications, and many others.

The typical workflow involves mapping these computational graphs to hardware resources such as GPUs and CPUs. However, before this hardware mapping occurs, there exists an opportunity to optimize the graph structure itself. This optimization can lead to significant performance improvements by reducing computational overhead, memory usage, and execution time.

The graphs exhibit both parallel and sequential computation patterns. Understanding these patterns is crucial for identifying optimization opportunities that can simplify the graph structure while maintaining mathematical equivalence.

Basic Graph Optimization Strategies

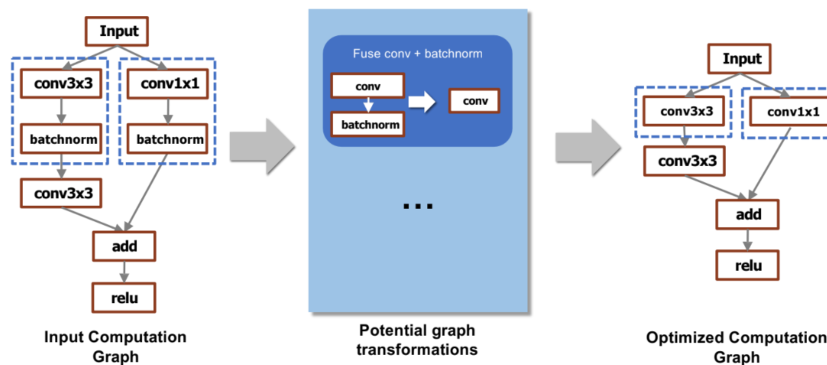

Figure 4: The computational graph optimization example showing potential transformation fusing conv + batchnorm [Diagram from MLSys-Intro PPT, Slide 7]

Graph-level optimizations involve various transformation techniques that can dramatically improve performance. One common scenario involves a computational graph with convolution layers followed by batch normalization operations. Through optimization, these separate operations can be fused together to create a simpler, more efficient graph structure.

Further optimization opportunities arise when examining the types of convolution operations present in the graph. For instance, when a graph contains both 1x1 convolutions and 3x3 convolutions, it may be beneficial to standardize these operations. Converting everything to 3x3 convolutions can enable more efficient computation since the same input can be processed by the same operator type. Alternatively, splitting operations might prove more cost-effective depending on the specific computational requirements and hardware characteristics.

Example: Fusing Convolution and Batch Normalization

Batch normalization serves to normalize the output of the preceding layer and then applies learned transformations such as scaling and shifting. During training, the batch normalization layer maintains running statistics like mean and variance, which are continuously updated. However, the key insight for optimization comes during the inference phase.

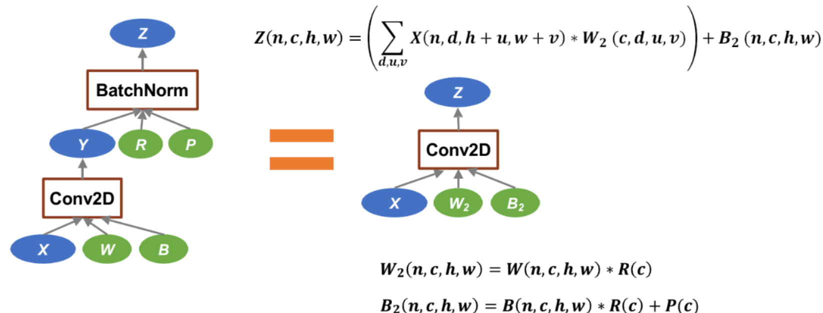

Figure 5: The computational graph visualizing the fusion of convolution and batchnorm fusion during inference. [Diagram from MLSys-Intro PPT, Slide 9]

During inference, the batch normalization parameters become fixed constants since they are derived from the completed training process. These include the running mean, running variance, and the learned scale and shift parameters (typically denoted as W, B, R, and P). Since these values are now constants rather than variables, the mathematical operations can be algebraically combined with the preceding convolution operation.

This fusion eliminates the need for separate normalization steps during inference. Instead of performing convolution followed by batch normalization as two distinct operations, the parameters can be pre-computed and combined into a single convolution operation with modified weights and biases. This reduces both computational overhead and memory access patterns, leading to improved performance.

Rule-Based Graph Transformation Systems

Current graph optimization systems primarily rely on rule-based approaches to perform these transformations. Systems like TensorFlow implement extensive rule databases that encode expert knowledge about beneficial graph transformations.

The rule-based approach works by analyzing the structure of the computational graph and identifying patterns that match predefined optimization rules. When a match is found, the system applies the corresponding transformation to produce a simplified or more efficient graph structure. This process continues iteratively until no more applicable rules can be found.

Current State-of-the-Art: TensorFlow's Rule-Based Optimizer

TensorFlow represents the current state-of-the-art in rule-based graph optimization, incorporating approximately 200 optimization rules. This extensive rule base has been developed through encoding expert knowledge about effective graph transformations.

The optimizer makes decisions based on the structure of the computational graph and the types of operators present. It systematically examines the graph for patterns that match its rule database and applies appropriate transformations. While this approach has proven effective in many scenarios, it faces significant scalability challenges as the complexity and diversity of ML models continue to grow.

The rule-based approach, while representing current best practice, has inherent limitations that become

more apparent as model architectures and hardware platforms diversify. The static nature of rule-based systems means they cannot adapt to new patterns or optimization opportunities that weren't anticipated during their development.

Limitations of Rule-Based Optimization Approaches

The limitations of rule-based optimization systems become apparent when examining real-world deployment scenarios. Google's XLA (Accelerated Linear Algebra) system serves as a representative example of these challenges.

XLA was designed with the goal of improving execution speed and reducing memory usage through aggressive graph optimization. However, practical deployment has revealed several critical limitations that highlight the broader challenges facing rule-based optimization approaches.

Robustness Challenges

The robustness problem manifests in the inability of rule-based systems to generalize across different model architectures and hardware platforms. There is no generalized technique that can guarantee positive results across all possible scenarios.

Scalability Issues

The scalability challenge relates to the fundamental question of rule completeness. With approximately 200 rules in systems like TensorFlow, it remains uncertain whether this represents a complete or even sufficient set of optimizations. As new operators are developed and novel architectures are created, the rule base must be continuously expanded.

Performance Limitations

Unlike data-driven or machine learning-based optimizers, rule-based systems may miss subtle optimization opportunities that could provide significant benefits for specific model and hardware combinations. The static nature of rules means they cannot adapt to the nuanced performance characteristics of different hardware platforms or the specific computational patterns of individual models.

Example: ResNet Optimization and Runtime Dependencies

The key insight from the ResNet example is that many optimization decisions cannot be made at compile time or during static analysis. Instead, they depend on factors that are only known at runtime, such as available memory, current system load, and specific hardware characteristics. This creates a fundamental challenge for static rule-based systems that must make optimization decisions without access to runtime information.

The performance results demonstrate this challenge clearly: the same optimization sequence that provides a 30% performance improvement on one GPU architecture (V100) actually degrades performance by 10% on another (K80).

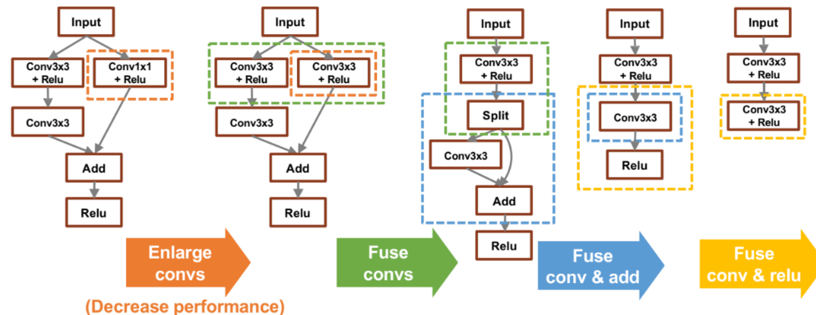

Figure 6: The computational graph showing performance of graph optimization with help of ResNet architecture. [Diagram from MLSys-Intro PPT, Slide 17]

The Combinatorial Explosion Problem

The fundamental challenge in graph optimization can be expressed as a combinatorial explosion problem. The space of possible optimizations is determined by the interaction of three major factors:

Graph Optimizations = ML Operators × Graph Architectures × Hardware Backends

Each of these dimensions contributes exponentially to the complexity of the optimization space. ML operators continue to evolve with new mathematical operations being developed for emerging AI techniques.

The combinatorial nature of this problem means that the number of possible optimization scenarios grows exponentially with each new operator, architecture pattern, or hardware platform. This mathematical reality makes it impractical to address the optimization problem through manual rule creation alone.

Hardware Backend Diversity

The hardware backend dimension of this problem is particularly challenging due to the rapid pace of innovation in AI accelerators. Different vendors implement fundamentally different approaches to high-performance computing. For example, NVIDIA's approach includes technologies like NVLink for fast transfer between CPU memory and GPU memory, while AMD implements different interconnect technologies with distinct performance characteristics.

Automated Graph Optimization

The ultimate goal of current open research in this area is to develop a generalized solution that can automatically adapt to the combinatorial complexity of modern ML optimization challenges. This represents a fundamental shift from manually crafted rules to automated optimization discovery and application.

Architecture of Automated Graph Optimizers

The structure of next-generation automated graph optimizers involves three key components working in coordination:

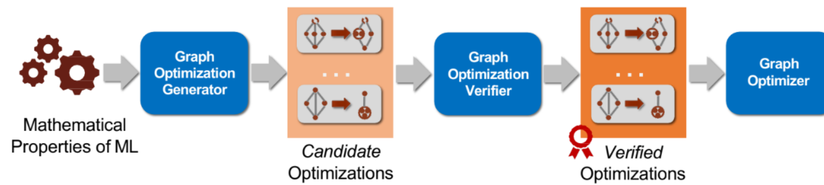

Figure 7: The step-by-step architecture for automated graph [Diagram from MLSys-Intro PPT, Slide 19]

Optimization Generation

The first component analyzes the structure of computational graphs and automatically generates potential optimization candidates.

Verification and Validation

The second component serves as a verification system that ensures proposed optimizations maintain output equivalence with the original graph.

Optimization Selection and Application

The final component takes the set of verified optimization candidates. Then we have the final graph optimizer.

Layer 3: Parallelization

Parallelization is the third layer, sitting between Graph-Level Optimization and Kernel Generation. It addresses how to distribute ML training across multiple computing devices to improve performance and handle larger models.

Three Stages of ML Training:

Training is typically done using Stochastic Gradient Descent (SGD), which has three core steps repeated over many iterations:

1. **Forward Propagation**
   Forward propagation is the process of passing input data through the network layers to obtain an output which is also called a prediction or inference.

   - Process a batch of input data by passing it through the model layers.

- Each layer applies a series of mathematical operations (e.g., linear transformations, activations) to transform the input.

- The final layer produces the model's prediction as the output of the forward pass.

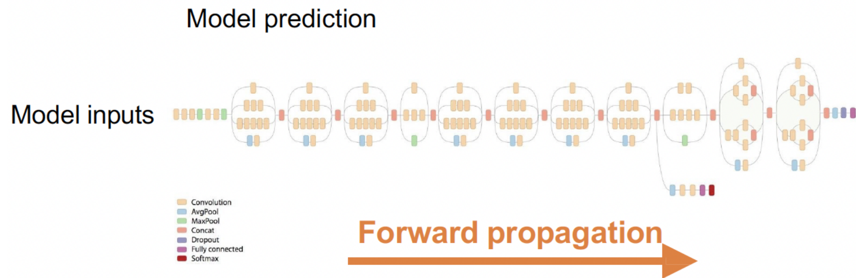

Figure: Forward Propagation [Diagram from MLSys-Intro PPT, Slide 21]

2. Backward Propagation

- The model is executed in reverse, starting from the output layer, to calculate how much each parameter contributed to the prediction error.
- Gradients (partial derivatives of the loss with respect to each parameter) are computed for all trainable weights.
- This process uses automatic differentiation, specifically backpropagation, to efficiently compute these gradients across all layers of the model.

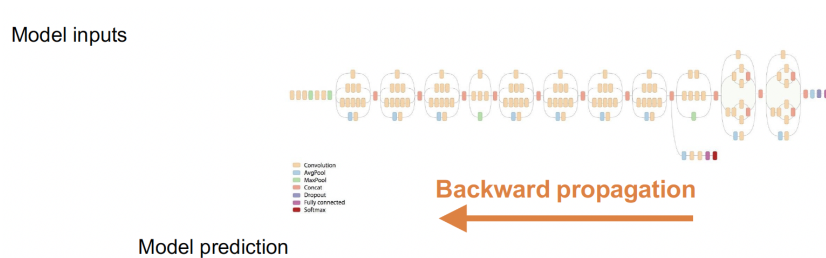

Figure: Backward Propagation [Diagram from MLSys-Intro PPT, Slide 22]

3. Weight Update
Computed gradients are used to update model weights:

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

where γ is the learning rate and n is the batch size.

Data Parallelism:

Data parallelism speeds up the training of machine learning models by distributing the workload across multiple GPUs. It involves splitting the input data and processing each subset in parallel, while synchronizing the model updates across devices.

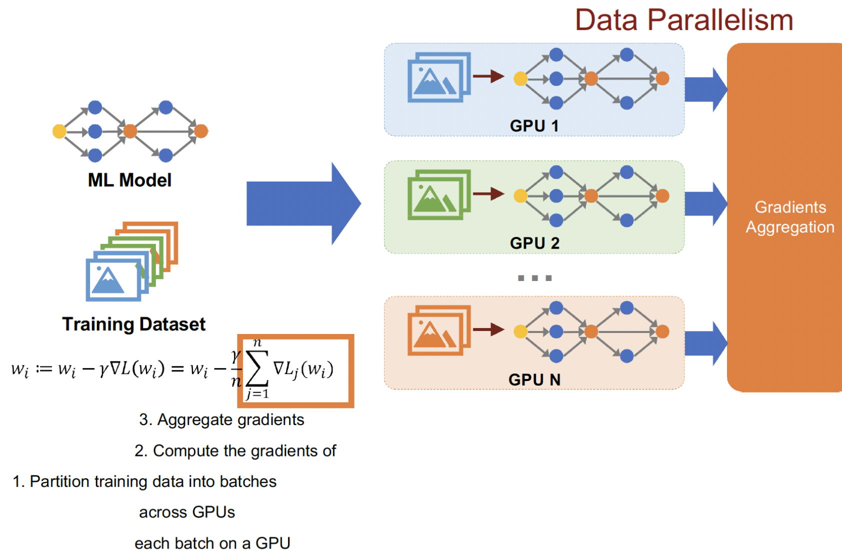

Figure: Data Parallelism [Diagram from MLSys-Intro PPT, Slide 25]

1. **Partition the training dataset into smaller batches.**
   - Each batch is assigned to a separate GPU.
2. **Parallel forward and backward passes:**
   - All GPUs use the same model architecture and weights.
   - Each GPU performs a forward pass on its local batch and computes the corresponding gradients via backward propagation.
3. **Aggregate gradients across GPUs:**
   - Gradients from all GPUs are collected and averaged.
   - This ensures consistency in parameter updates.
4. **Update model weights:**
   - A global weight update is applied using the aggregated gradients.
   - For example:

$$w_i := w_i - \gamma \cdot \frac{1}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

where $\nabla L_j(w_i)$ is the gradient of the loss on GPU j, and γ is the learning rate.

Model Parallelism:

Model parallelism is used when the model is too large to fit on a single GPU. Instead of splitting the data, we split the model itself across multiple devices.

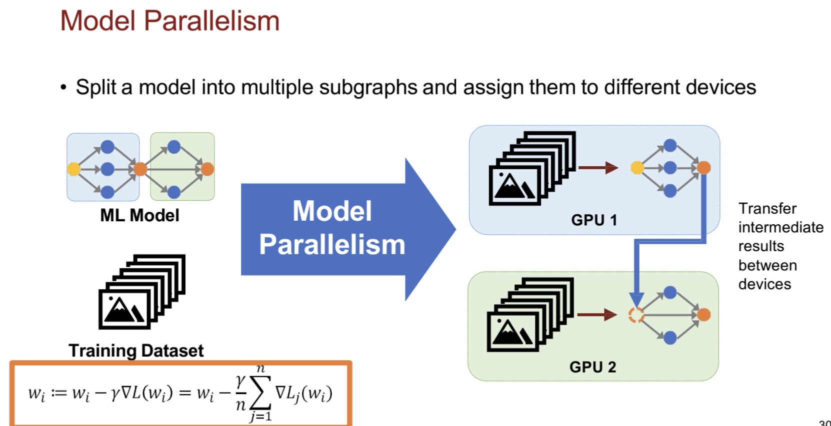

Figure: Model Parallelism [Diagram from MLSys-Intro PPT, Slide 30]

Steps in Model Parallel Training

1. **Split the model into subgraphs:**
   - Different layers of the model are assigned to different GPUs.
2. **Each GPU processes a portion of the model:**
   - For example, GPU 1 handles the earlier layers, while GPU 2 handles the later layers.
3. **Transfer intermediate activations between GPUs:**
   - After GPU 1 processes its part of the model, the intermediate results are passed to GPU 2 to continue the forward pass.
   - Similarly, during the backward pass, gradients flow in reverse across devices.
4. **Synchronize parameter updates if needed:**
   - Each subgraph updates only its own weights, but proper coordination is needed to ensure correctness.

The weight update rule is:

$$w_i := w_i - \gamma \cdot \frac{1}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

where $\nabla L_j(w_i)$ is the gradient of the loss with respect to parameter w_i computed on GPU j, and γ is the learning rate.