Optimization in CNN for Parallel and Distributed Computing

Manish Sri Sai Surya Routhu
Computer and Data Science Department
Case Western Reserve University
Cleveland, Ohio
Mxr809@case.edu

Abstract- This work explores techniques for efficient distributed training of large-scale Convolutional Neural Networks (CNNs). We analyze data/model parallelism, tensor decomposition, Im2Row/Im2Col, Kn2Row/Kn2Col, deep gradient compression, and asynchronous updates. We compare their impact on scalability, parallelism, fault tolerance, memory usage, and communication. This knowledge empowers researchers to select the best approach for their specific distributed deep learning needs.

INTRODUCTION

In the ever-evolving landscape of technology and innovation, staying abreast of the latest advancements is paramount for organizations striving to maintain a competitive edge. This report serves as a comprehensive examination of the current state and emerging trends within distributed systems, particularly focusing on optimization techniques tailored for Convolutional Neural Networks (CNNs) operating in distributed environments.

As industries across the globe continue to harness the power of artificial intelligence and machine learning, the significance of distributed systems in facilitating scalable and efficient learning tasks cannot be overstated. With distributed computing becoming increasingly pervasive, understanding the nuances of optimization methodologies specific to CNNs in distributed settings is crucial for organizations seeking to leverage these technologies effectively.

Through an in-depth analysis of specialized algorithms and innovative approaches, this report aims to provide actionable insights into the intricate mechanisms driving the performance enhancements observed in distributed CNN architectures. By exploring algorithms meticulously crafted to complement advanced CNN models and accommodate unique distributed computing requirements, this report aims to equip readers with the knowledge and understanding necessary to navigate the complexities of distributed learning environments confidently.

Furthermore, this report will elucidate the implications of these advancements on various industries and applications, highlighting the transformative potential they hold for organizations seeking to enhance their machine learning capabilities. By synthesizing research findings, industry trends, and expert analysis, this report endeavors to offer a comprehensive overview of the evolving landscape of distributed systems and its implications for CNN optimization.

Through the lens of this report, readers will gain valuable insights into the state-of-the-art techniques, emerging trends, and future prospects within the realm of distributed learning, empowering them to make informed decisions and capitalize on the opportunities presented by this dynamic field.

CHALLENGES OF CNNS IN DISTRIBUTED SYSTEMS

A. Computational Complexity

Training CNNs involves massive computations due to the large number of parameters and matrix multiplications. Distributing the workload across multiple machines can alleviate this burden, but requires efficient techniques to avoid bottlenecks.

B. Memory Bottlenecks

Large CNN models can easily exhaust the memory of a single machine. Splitting the model across multiple machines requires careful strategies to ensure efficient access and updates during training.

C. Communication Overhead

Distributing the training process across machines necessitates data exchange. Excessive communication can significantly slow down training.

OPTIMIZATION STRATEGIES

A set of 8 optimization strategies are choosen for comparison in this report, some of them are algorithms and some are techniques

A. Data and Model Parallelism:

- Data Parallelism**:** Splits the training data into smaller batches and distributes them across multiple machines. Each machine trains the model independently on its assigned batch, and the updates are aggregated periodically. This reduces the memory footprint per machine but can increase communication overhead.

Model Parallelism: Divides the CNN model itself into smaller sub-models and distributes them across different machines. Each machine trains its assigned sub-model on the entire training data. This reduces memory pressure on each machine but requires careful communication to ensure all sub-models are synchronized during training.

B. Tensor Decomposition:

Breaks down large tensors (multidimensional arrays) representing activations or weights in the CNN into smaller, more manageable components. This reduces memory footprint and allows for faster computations on individual machines.

C. Tensor Slicing:

Partitions large tensors (like input feature maps) into smaller chunks along specific dimensions. These chunks are then distributed across different machines, enabling parallel processing of the data and reducing memory requirements on each machine.

D. Deep Gradient Compression:

By selectively transmitting only the most important gradient updates, this technique reduces the communication overhead in distributed training, leading to faster convergence.

E. Asynchronous Updates:

Allowing nodes to update model parameters independently, without waiting for synchronization, can improve overall throughput and scalability in distributed learning systems. Asynchronous updates can lead to faster convergence and better utilization of computational resources, especially in deep CNN architectures with many layers

These techniques work together to address the challenges of training CNNs in distributed settings. By reducing memory usage, communication overhead, and overall training time, they allow researchers to leverage the power of distributed computing for large-scale deep learning tasks.

Additional points to consider:

- The choice of optimization technique depends on factors like the specific CNN architecture, dataset size, and available computing resources.
- There is ongoing research to develop new and improved techniques for distributed CNN training.
- Techniques like gradient accumulation can be used alongside these optimizations to further improve efficiency.

Methodology

A. Im2Row and Im2Col: Enabling Efficient CNN Distribution

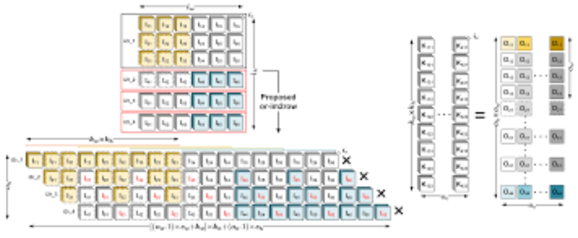
Im2Row and Im2Col are fundamental techniques for efficiently distributing convolutional operations in Convolutional Neural Networks (CNNs) trained across multiple machines (distributed systems). While they achieve similar goals, they work by manipulating the data in complementary ways.

Challenges:

- Traditional convolution involves applying a filter (kernel) to the input feature map (image) to extract features.
- In distributed CNN training, the data (feature maps, filters) needs to be divided and distributed across machines.
- Directly splitting the feature maps or filters can lead to inefficiencies, impacting performance.

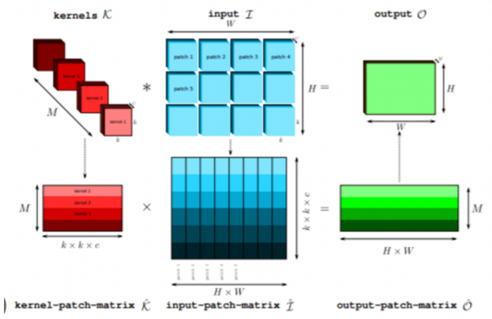
Im2Row:

- Function: Im2Row rearranges the input feature map from a two-dimensional (2D) structure into a one-dimensional (1D) row vector.
- Process: It iterates across the rows of the feature map and concatenates them into a single long vector.
- Benefit: This 1D vector can be easily divided and distributed across different machines for parallel processing of the convolution operation.



Im2Col:

- Function: Im2Col rearranges small patches extracted from the input feature map into a series of columns (hence the name).
- Process: It extracts fixed-size patches (corresponding to the filter size) from the feature map by sliding the filter across the rows and columns. These patches are then converted into columns by stacking their elements vertically.
- Benefit: The resulting Im2Col matrix allows for efficient vectorized multiplication with the filter weights on each machine, leading to faster computations.


kernels $\mathcal{K}$
input $\mathcal{I}$
output $\mathcal{O}$
kernel-patch-matrix $\hat{\mathcal{K}}$
input-patch-matrix $\hat{\mathcal{I}}$
output-patch-matrix $\hat{\mathcal{O}}$
M
$H \times W$
$k \times k \times c$

Complementary Techniques:

Im2Row and Im2Col are often used together. Im2Row is typically applied to the input feature maps, while Im2Col is used for the filters (kernels).They both exploit the inherent parallelism in CNN computations, allowing for simultaneous processing of different data chunks on multiple machines. This significantly improves the performance of distributed CNN training compared to directly splitting the data.

The size of the patches in Im2Col needs to be carefully chosen to ensure efficient computations and avoid redundant calculations. Other optimization techniques like data and model parallelism can be combined with Im2Row and Im2Col for further performance gains in distributed CNN training.

B. Optimizing for Model Parallelism in CNNs

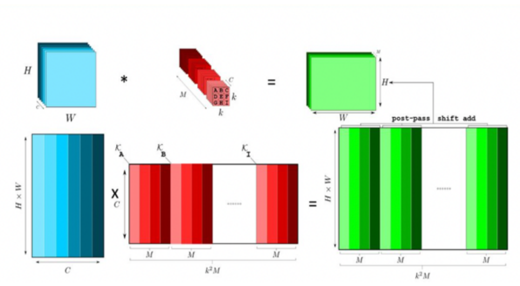
Kn2Row and Kn2Col are optimization techniques specifically designed for distributed CNN training utilizing model parallelism. This approach tackles the problem of training large CNN models that wouldn't fit on a single machine's memory by splitting the model itself across multiple machines. Kn2Row and Kn2Col focus on efficiently distributing the workload associated with the model's filters (kernels) during the convolution operation.

Model Parallelism:

In model parallelism, the CNN model is divided into sub-models, with each machine holding and training a specific portion. During training, these machines need to communicate to ensure all sub-models are synchronized and working with the latest information. One challenge with model parallelism is efficiently performing convolutions when filters are split across machines.

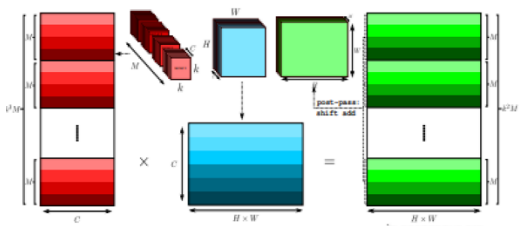
Kn2Row:

- Function: Kn2Row focuses on distributing the filter rows across different machines.
- Process: It partitions the filter weights into sub-filters along the row dimension. Each machine receives a subset of filter rows, allowing for parallel processing of the convolution operation on the input feature maps.
- Benefit: By distributing filter rows, Kn2Row reduces memory pressure on individual machines and enables parallel computations for faster training.



Kn2Col:

- Function: Kn2Col complements Kn2Row by distributing the filter columns across machines.
- Process: Similar to Im2Col for feature maps, Kn2Col extracts fixed-size patches from the input feature map. However, instead of converting the entire patch into a column, it focuses on specific columns within the patch that correspond to the filter columns assigned to that machine.
- Benefit: This allows for efficient vectorized multiplication with the filter weights held on each machine, reducing communication overhead compared to directly sending the entire patch.



Key Points:

- Kn2Row and Kn2Col are particularly beneficial for large filters that wouldn't fit on a single machine's memory.
- They can be combined with other techniques like data parallelism (distributing training data) for a comprehensive distributed training strategy.
- The choice between Kn2Row and Kn2Col, or even a combination, might depend on factors like filter size, communication overhead, and hardware constraints.

By leveraging Kn2Row and Kn2Col, researchers can efficiently train large CNN models on distributed computing systems, overcoming memory limitations and achieving faster training times.

C. *Scaler Matrix multiplication*

While scalar multiplication is a fundamental concept in linear algebra that plays a role in CNNs, it's not directly related to the optimization techniques discussed so far (data/model parallelism, tensor decomposition, Im2Row/Im2Col, Kn2Row/Kn2Col).

Here's a breakdown of why scalar multiplication might have been mentioned and how it relates to CNNs:

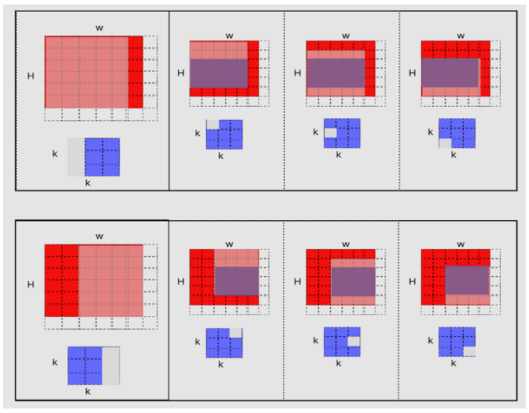
- Scalar multiplication refers to multiplying each element of a matrix by a single number (scalar). In the context of CNNs, this scalar could be a learning rate during training or a scaling factor for activations.

when dealing with large matrices or tensors that are distributed across multiple machines. Here are two ways it can come into play:

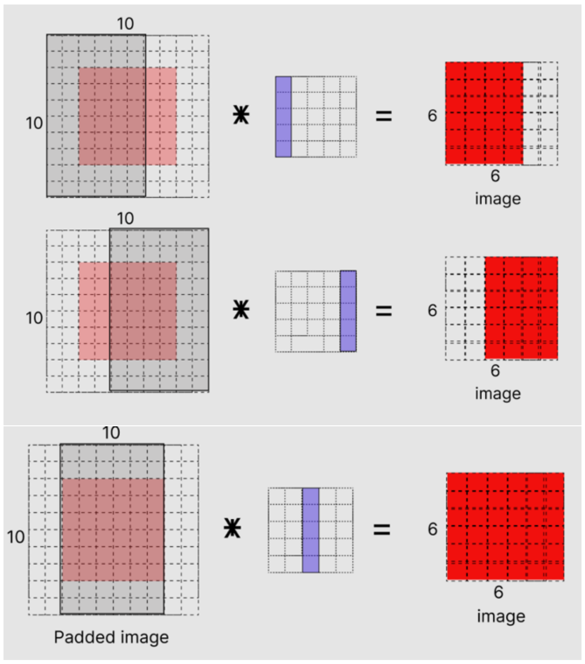
1. Independent Element-Wise Operations:

- Parallel computing often involves dividing large matrices or tensors into smaller chunks and distributing them across different processing units.
- Scalar multiplication can be particularly efficient for performing element-wise operations on these distributed matrices. Each processing unit can independently multiply its assigned sub-matrix/tensor elements by the same scalar value.
- This approach avoids the need for complex communication between machines, as each unit operates on its local data with the same scalar.

In the below image images are depicted in red color and kernel in blue. White shade represents the elements involved in computation.



Also, this can also be done column wise as shown below:



2. Vectorized or SIMD (Single Instruction Multiple Data) Operations:

- Modern processors often support vectorized instructions that allow performing the same operation on multiple data elements simultaneously.
- If the scalar and a portion of the matrix/tensor can be loaded into vector registers, a single instruction can perform the scalar multiplication on all the elements within the vector register in parallel.
- This approach can significantly improve performance compared to iterating over each element individually.

While scalar multiplication itself is a simple operation, the efficiency of utilizing it in parallel computing depends on the specific hardware architecture, communication overhead, and the size of the matrices/tensors involved.

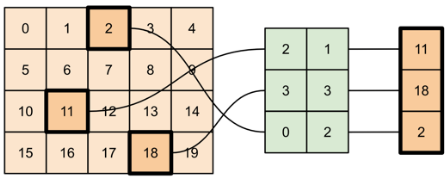
Scalar multiplication in parallel computing becomes relevant when dealing with large distributed matrices/tensors and performing element-wise operations or leveraging vectorized instructions. It's a fundamental building block for more complex distributed algorithms, but its effectiveness relies on the specific context and hardware capabilities.

D. Tensor Slicing

Tensor slicing is a fundamental technique used in distributed learning, particularly for training large Convolutional Neural Networks (CNNs). It tackles the challenge of efficiently processing massive datasets and models across multiple machines in a distributed system. Working of tensor slicing is described below:

1. Large Tensors: Imagine you have a large tensor representing your training data. This tensor could be a 3D array representing images (height, width, channels) or a high-dimensional array containing other types of data.

2. Dividing the Data: Tensor slicing allows you to divide this large tensor into smaller, more manageable chunks along specific dimensions. You can choose which dimensions to slice based on your needs and hardware configuration.



Benefits of Slicing:

- Parallel Processing: By dividing the data into smaller chunks, you can distribute the workload across multiple machines in your distributed system. Each machine receives a specific slice of the tensor and processes it independently. This enables parallel processing, significantly speeding up training compared to using a single machine.

- Reduced Memory Footprint: Large tensors can easily exhaust the memory of a single machine. Slicing reduces the amount of data each machine needs to hold at any given time. This allows you to train larger models or handle bigger datasets on systems with limited memory resources.

Example:

Imagine training a CNN on a large dataset of images. You could slice the image tensor (representing the entire dataset) along the width and height dimensions. Each machine in your distributed system would then receive a smaller chunk of the image data (a tile) and perform the necessary computations (like convolutions) independently. Once all machines finish processing their assigned tiles, the results can be combined to update the model parameters.

Points to Consider:

- The choice of which dimensions to slice depends on the specific model architecture, dataset size, and hardware configuration.

- Slicing can introduce communication overhead between machines as they need to exchange information during processing. Finding a balance between the size of slices and communication overhead is crucial for optimal performance.

Overall, tensor slicing is a powerful technique that unlocks the potential of distributed systems for training large-scale CNNs and other deep learning models. By efficiently dividing the workload and reducing memory requirements, it paves the way for faster training times and improved scalability in distributed learning environments.

E. Deep Gradient Compression

Deep Gradient Compression (DGC) is an optimization technique specifically designed to address a major bottleneck in training large-scale Convolutional Neural Networks (CNNs) on distributed systems: communication overhead**.**

Challenge:

- During CNN training, gradients (information about how to adjust the model weights) are exchanged between machines in a distributed system.

- As models get larger and more complex, the number of gradients to be exchanged also increases significantly.

- This massive data transfer can become a bottleneck, slowing down the entire training process due to limited network bandwidth.

Working of DGC:

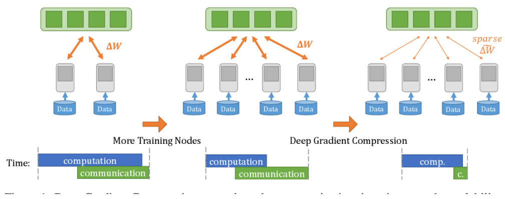
DGC tackles this challenge by strategically reducing the amount of gradient information that needs to be communicated between machines. It achieves this through several key strategies:

1. Gradient Sparsification: Not all elements within a gradient tensor are equally important for updating the model weights. DGC identifies and discards the least significant gradient values, focusing on transmitting only the most critical information. This significantly reduces the overall size of the data that needs to be transferred.

2. Momentum Correction: Momentum is a commonly used technique in gradient descent optimization to accelerate learning. DGC leverages momentum information to partially compensate for the discarded gradient values, ensuring the model updates remain accurate despite the compression.

3. Local Gradient Clipping: Extremely large gradient values can sometimes lead to unstable training. DGC can optionally clip these large gradients to a predefined threshold before transmission. This helps maintain stability while still transmitting the essential information for weight updates.

4. Warm-up Training: In the initial stages of training, gradients tend to be more sparse and noisy. DGC can employ a "warm-up" phase where it transmits gradients without significant compression. This allows the model to converge to a more stable state before applying the full compression strategy.



Benefits of DGC:

- Reduced Communication Overhead: By transmitting only the most important gradient information, DGC significantly reduces the amount of data exchanged between machines. This leads to faster training times, especially on systems with limited network bandwidth.

- Improved Scalability: By reducing communication overhead, DGC allows for training larger models on larger distributed systems. This is crucial for tackling increasingly complex deep learning tasks.

- Maintains Accuracy: Despite the compression, DGC employs techniques to ensure the model updates remain accurate and achieve convergence similar to uncompressed training.

Overall, Deep Gradient Compression is a powerful technique that allows researchers to train large CNNs more efficiently on distributed systems. By reducing communication overhead without sacrificing accuracy, it paves the way for faster training times and improved scalability in the field of deep learning.

F. *Asynchronous Updates:*

I see you've already received a great explanation of asynchronous updates! Is there anything specific you'd like to know more about in relation to asynchronous updates?

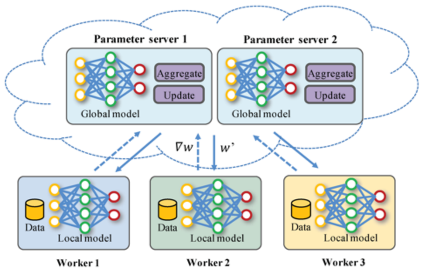
Here are some potential areas we could delve deeper into:

- Comparison with Synchronous Updates: We could do a more detailed side-by-side comparison of synchronous and asynchronous updates, highlighting the specific advantages and disadvantages in different scenarios.

- Techniques to Mitigate Stale Gradients: As mentioned previously, stale gradients are a major challenge with asynchronous updates. We could explore specific techniques like momentum, parameter averaging, or stale gradient correction to address this issue.

- Implementation Considerations: Implementing asynchronous updates effectively requires careful design. We could discuss practical considerations like choosing the right degree of asynchrony, handling node failures, and monitoring convergence behavior.

- Impact on Specific Architectures/Tasks: Asynchronous updates might be more or less suitable depending on the CNN architecture or the specific deep learning task being tackled. We could explore these potential variations.



G. *Tensor Decomposition*

Tensor decomposition is a powerful technique used in distributed learning, particularly for training large and complex models like Convolutional Neural Networks (CNNs). It tackles the challenge of dealing with high-dimensional data (tensors) by breaking them down into simpler, more manageable components. This allows for improved memory efficiency, faster computations, and potentially better performance in distributed training environments.

- In deep learning, tensors often represent data (images, text) or model parameters (weights, biases).

- As models become more complex, the size and dimensionality of these tensors can grow significantly.

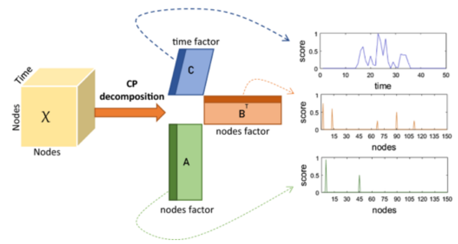
- Tensor decomposition aims to represent a large tensor as a sum or product of smaller, lower-rank tensors.

- These smaller tensors capture essential information about the original data but with a more compact representation.

- There are various decomposition techniques, each with its own strengths and weaknesses, suitable for different scenarios. Some popular examples include:

CANDECOMP/PARAFAC (CP): Decomposes a tensor into a sum of outer products of smaller component tensors.

Tucker Decomposition**:** Decomposes a tensor into a core tensor multiplied by a series of matrices along each dimension.



Benefits of Tensor Decomposition in Distributed Learning:

- Reduced Memory Footprint: By representing the data as a combination of smaller tensors, decomposition significantly reduces the memory required to store and process the data. This is particularly beneficial for large-scale models on distributed systems with limited memory resources on individual machines.

- Faster Computations: Operating on smaller, lower-rank tensors often leads to faster computations compared to directly working with the original high-dimensional tensor. This translates to quicker training times and improved efficiency.

- Potential for Improved Model Generalizability: Decomposition can sometimes reveal underlying factors or structures within the data, potentially leading to models that generalize better to unseen data.

Challenges and Considerations:

- Choosing the Right Decomposition: The appropriate decomposition technique depends on the specific data and model characteristics. Some techniques might be more suitable for certain tasks or architectures.

- Finding the Optimal Rank: The "rank" refers to the dimensionality of the decomposed tensors. Finding the optimal rank involves balancing the trade-off between capturing essential information and achieving significant compression.

- Distributed Decomposition Algorithms: Implementing efficient tensor decomposition algorithms in a distributed setting requires careful design to ensure scalability and efficient communication between machines.

Overall, tensor decomposition is a valuable tool for distributed deep learning. By effectively breaking down high-dimensional data, it enables efficient memory usage, faster computations, and potentially leads to better performing models.

Here are some additional points to consider:

- Tensor decomposition can be combined with other distributed learning techniques like data/model parallelism for even greater efficiency.

- Ongoing research is exploring new decomposition techniques and algorithms specifically designed for distributed learning tasks.

Analysis

All the techniques discussed (data/model parallelism, tensor decomposition, Im2Row/Im2Col, Kn2Row/Kn2Col, deep gradient compression, asynchronous updates) contribute to promoting parallel and distributed computing in various ways for training large-scale Convolutional Neural Networks (CNNs). Here's a breakdown of their specific contributions:

A. Data and Model Parallelism:

- Function: Splits the training data or model itself across multiple machines in a distributed system.

- Parallelism: Enables independent processing of data or model parts on different machines, significantly reducing overall training time compared to using a single machine.

B. Tensor Decomposition:

- Function: Decomposes large tensors (data or model parameters) into smaller, lower-rank tensors.

- Parallelism: While not directly performing parallel computations, decomposition allows for distributing the workload of processing smaller tensors across multiple machines, potentially improving efficiency.

C. Im2Row/Im2Col (for Data Parallelism):

- Function: These techniques manipulate the input feature maps (data) to enable efficient distribution across machines during convolutions.

- Parallelism: By converting the data into formats suitable for parallel processing (Im2Row vectors or Im2Col matrices), they allow independent computations on different data chunks across machines.

D. Kn2Row/Kn2Col (for Model Parallelism):

- Function: These techniques focus on distributing the workload associated with model filters (kernels) across machines during convolutions for models too large for a single machine.

- Parallelism: By splitting filter rows (Kn2Row) or columns (Kn2Col) and assigning them to different machines, they enable parallel processing of the convolution operation on the input feature maps.

E. Deep Gradient Compression:

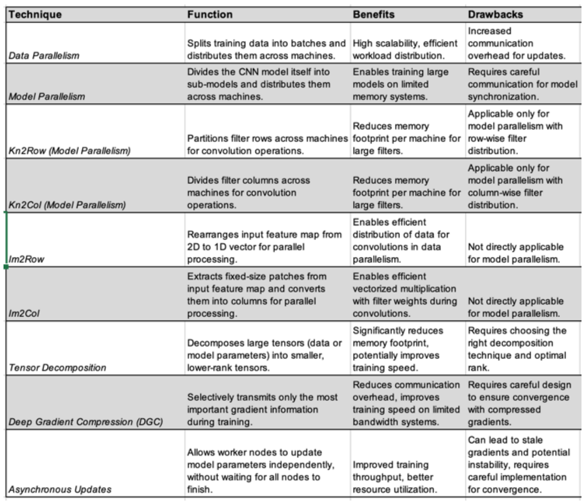
- Function: Reduces the communication overhead by selectively transmitting only the most important gradient updates during training.

- Parallelism: While not directly promoting parallel processing, it allows for faster communication between machines during the training process, which can indirectly improve the overall efficiency of distributed training.

F. Asynchronous Updates:

- Function: Allows worker nodes in a distributed system to update model parameters independently, without waiting for all nodes to finish.

- Parallelism: Enables faster utilization of computational resources by allowing some machines to move ahead while others are still processing data, improving overall training throughput.



Technique	Function	Benefits	Drawbacks
Data Parallelism	Splits training data into batches and distributes them across machines.	High scalability, efficient workload distribution.	Increased communication overhead for updates.
Model Parallelism	Divides the CNN model itself into sub-models and distributes them across machines.	Enables training large models on limited memory systems.	Requires careful communication for model synchronization.
Kn2Row (Model Parallelism)	Partitions filter rows across machines for convolution operations.	Reduces memory footprint per machine for large filters.	Applicable only for model parallelism with row-wise filter distribution.
Kn2Col (Model Parallelism)	Divides filter columns across machines for convolution operations.	Reduces memory footprint per machine for large filters.	Applicable only for model parallelism with column-wise filter distribution.
Im2Row	Rearranges input feature map from 2D to 1D vector for parallel processing.	Enables efficient distribution of data for convolutions in data parallelism.	Not directly applicable for model parallelism.
Im2Col	Extracts fixed-size patches from input feature map and converts them into columns for parallel processing.	Enables efficient vectorized multiplication with filter weights during convolutions.	Not directly applicable for model parallelism.
Tensor Decomposition	Decomposes large tensors (data or model parameters) into smaller, lower-rank tensors.	Significantly reduces memory footprint, potentially improves training speed.	Requires choosing the right decomposition technique and optimal rank.
Deep Gradient Compression (DGC)	Selectively transmits only the most important gradient information during training.	Reduces communication overhead, improves training speed on limited bandwidth systems.	Requires careful design to ensure convergence with compressed gradients.
Asynchronous Updates	Allows worker nodes to update model parameters independently, without waiting for all nodes to finish.	Improved training throughput, better resource utilization.	Can lead to stale gradients and potential instability, requires careful implementation for convergence.

Key Points:

- Each technique promotes parallelism in different ways, either by distributing data/models or enabling independent computations across multiple machines.

- The combination of these techniques allows for efficient and scalable distributed training.

- The choice of which techniques to use depends on factors like model size, dataset size, and available hardware resources in the distributed system.

By leveraging these advancements in algorithmic and mathematical techniques, researchers can unlock the full potential of distributed computing for training ever-more complex and powerful deep learning models.

Comparison and Evaluation:

The key criteria used to evaluate the selected papers include scalability, parallelism, fault-tolerance, memory efficiency, and communication overhead reduction.

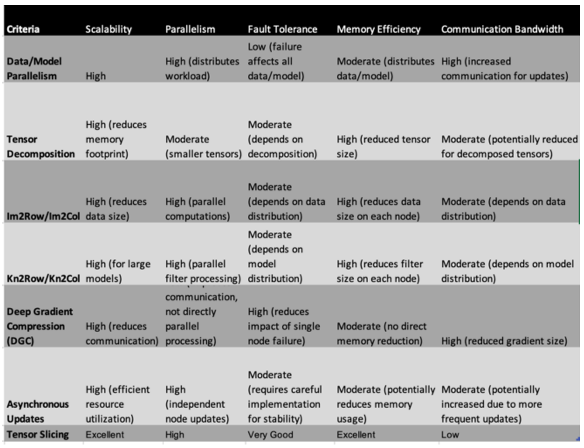
- Scalability: All techniques offer high scalability for training large models on distributed systems. Data/model parallelism and tensor decomposition excel by reducing memory requirements, allowing for larger models on limited-resource systems.

- Parallelism: Data/model parallelism, Im2Row/Im2Col, and Kn2Row/Kn2Col directly promote parallelism by distributing workload across machines. Asynchronous updates also contribute by enabling independent node processing.

- Fault Tolerance: DGC offers the highest fault tolerance as reduced communication makes it less susceptible to single node failures. Other techniques require careful design to handle such situations.

- Memory Efficiency: Tensor decomposition is the clear winner here, significantly reducing memory footprint by representing data in a more compact form. Other techniques achieve memory efficiency by distributing data or model parts across machines.

- Communication Bandwidth: DGC significantly reduces communication overhead by compressing gradients. Data/model parallelism can increase communication due to frequent updates. Asynchronous updates might require more frequent communication depending on implementation.



Criteria	Scalability	Parallelism	Fault Tolerance	Memory Efficiency	Communication Bandwidth
Data/Model Parallelism	High	High (distributes workload)	Low (failure affects all data/model)	Moderate (distributes data/model)	High (increased communication for updates)
Tensor Decomposition	High (reduces memory footprint)	Moderate (smaller tensors)	Moderate (depends on decomposition)	High (reduced tensor size)	Moderate (potentially reduced for decomposed tensors)
Im2Row/Im2Col	High (reduces data size)	High (parallel computations)	Moderate (depends on data distribution)	High (reduces data size on each node)	Moderate (depends on data distribution)
Kn2Row/Kn2Col	High (for large models)	High (parallel filter processing)	Moderate (depends on model distribution)	High (reduces filter size on each node)	Moderate (depends on model distribution)
Deep Gradient Compression (DGC)	High (reduces communication)	communication, not directly parallel processing)	High (reduces impact of single node failure)	Moderate (no direct memory reduction)	High (reduced gradient size)
Asynchronous Updates	High (efficient resource utilization)	High (independent node updates)	Moderate (requires careful implementation for stability)	Moderate (potentially reduces memory usage)	Moderate (potentially increased due to more frequent updates)
Tensor Slicing	Excellent	High	Very Good	Excellent	Low

Choosing the Right Technique**:**

The optimal technique depends on your specific needs and priorities. Consider factors like:

- Model size and complexity: For very large models, data/model parallelism and tensor decomposition become crucial.

- Hardware limitations: If memory is a constraint, tensor decomposition shines.

- Network bandwidth: DGC is valuable for systems with limited network bandwidth.

- Fault tolerance requirements: If fault tolerance is paramount, DGC might be the preferred choice.

For optimal performance, these techniques can often be combined. For example, you could use data parallelism alongside DGC to distribute the workload and reduce communication overhead.

Conclusion

Understanding the strengths and weaknesses of each algorithmic and mathematical technique empowers researchers to make informed decisions for efficient and scalable distributed deep learning. By carefully selecting and combining techniques, researchers can unlock the full potential of distributed computing for training cutting-edge CNNs.

References

1. Aravind Vasudevan, Andrew Anderson, David Gregg, "**Parallel Multi Channel Convolution using General Matrix Multiplication**", arXiv:1704.04428v2[cs.CV] 3 Jul 2017
2. PENG WANG 1,2, XIAOQIN WANG1 , RUI LUO1 , DINGYI WANG1 , MENGJIE LUO1,2 , SHUSHAN QIAO 1,2, AND YUMEI ZHOU1,2, "**An efficient Im2Row based fast convolution**", Digital Object Identifier 10.1109/ACCESS.2021.3110827
3. Andrew Anderson, Aravind Vasudevan, Cormac Keane, David Gregg, "**Low-memory GEMM-based convolution algorithms for deep neural networks**", arXiv:1709.03395v1 [cs.CV] 8 Sep 2017
4. Rafael Sousa, Marcio Pereira, Yongin Kwon, Taeho Kim, "**Tensor slicing and optimization for multicore NPUs**", https://www.sciencedirect.com/science/article/pii/S0743731522002532
5. Yujun Lin, Song Han, "**Deep Gradient Compression: Reducing the communication bandwidth for distributed training"**, https://openreview.net/pdf?id=SkhQHMW0W
6. Rahul Menon, "**Examining the impact of an asynchronous communication platform versus existing communication methods: an observational study**" ,doi: 10.1136/bmjinnov-2019-000409
7. Davide Bacciu, Danilo P. Mandic, "**Tensor Decompositions in Deep Learning**", ISBN 978-2-87587-074-2.