Technical Report

A Survey of Distributed Learning in Cloud, Mobile, and Edge Settings

Madison Threadgill
Andreas Gerstlauer

UT-CERC-24-01

May 1, 2024

Computer Engineering Research Center
Chandra Family Department of Electrical and Computer Engineering
The University of Texas at Austin

The University of Texas at Austin
Chandra Department of Electrical and Computer Engineering
Cockrell School of Engineering

A Survey of Distributed Learning in Cloud, Mobile, and Edge Settings

Madison Threadgill, Andreas Gerstlauer

Electrical and Computer Engineering
The University of Texas at Austin

Abstract

In the era of deep learning (DL), convolutional neural networks (CNNs), and large language models (LLMs), machine learning (ML) models are becoming increasingly complex, demanding significant computational resources for both inference and training stages. To address this challenge, distributed learning has emerged as a crucial approach, employing parallelization across various devices and environments. This survey explores the landscape of distributed learning, encompassing cloud and edge settings. We delve into the core concepts of data and model parallelism, examining how models are partitioned across different dimensions and layers to optimize resource utilization and performance. We analyze various partitioning schemes for different layer types, including fully connected, convolutional, and recurrent layers, highlighting the trade-offs between computational efficiency, communication overhead, and memory constraints. This survey provides valuable insights for future research and development in this rapidly evolving field by comparing and contrasting distributed learning approaches across diverse contexts.

Table of Contents

List of Tables . . . 3

List of Figures . . . 4

Chapter 1: Introduction . . . 5

Chapter 2: Data and Model Partitioning . . . 8

Chapter 3: Layer Partitioning . . . 13

3.1 Fully Connected Layers . . . 13

3.2 Convolutional Layers . . . 16

3.2.1 Feature Map Partitioning . . . 16

3.2.2 Channel, Filter, and Weight Partitioning . . . 18

3.3 Recurrent Layers . . . 20

3.3.1 Gate-Based Partitioning . . . 22

3.3.2 Weight-Based Partitioning . . . 23

3.3.3 Model Partitioning . . . 24

Chapter 4: Challenges and Future Directions . . . 25

Chapter 5: Summary and Conclusions . . . 27

Bibliography . . . 28

List of Tables

2.1 Model and data parallelism in ML networks. 10

3.1 Partitioning schemes for fully connected layers. 15
3.2 Partitioning schemes for convolutional neural networks. 17
3.3 Existing works on RNN partitioning. 22

List of Figures

2.1 Data and model parallelism in a neural network. 9

3.1 Fully connected layer parallelism. 14
3.2 Convolutional layer parallelism. 16
3.3 Shared data in distributed convolutional operations. 18
3.4 Long Short-Term Memory (LSTM) cell. 20

Chapter 1: Introduction

With the rise of deep learning, convolutional neural networks (CNNs), and large language models (LLMs), machine learning (ML) models are increasing in computational complexity during both the inference and training stages.

Parallelization methods have been introduced to overcome the computational cost associated with ML models. There are many different granularities as to which ML models can be parallelized. Fine-grained parallelism can occur in shared memory systems where operational or thread-level parallelism is exploited. This form of parallelism is largely well understood with standard approaches being implemented in most systems today. On the other hand, coarse-grained parallelism can be achieved by distributing the ML model across various devices. This form of parallelism introduces challenges as explicit partitioning of data and/or models must be maintained in a distributed memory fashion. Moreover, the partitioning method must also keep in mind the communication overhead between devices to maintain the performance of the system. Additionally, when implementing coarse-grained partitioning, multiple options arise when determining the system's architecture, such as involving the cloud for computational offloading or keeping all data on edge and/or mobile devices.

In the cloud, partitioning of an ML model is typically implemented to mitigate significant computational costs associated with training. The training process can be distributed across multiple CPU or GPU nodes in a cloud or data center cluster. By contrast, when moving computations towards the edge, where resources are limited, inference tasks are commonly used instead of training due to a decreased computational complexity. Moreover, implementing partitioning methods that specifically account for memory and communication impacts is crucial for fast processing of inference tasks.

Edge and mobile devices are resource constrained with limited computational and communication capabilities. Memory constraints make fitting an entire ML model on a single edge device often impossible. Hence, clusters of edge and mobile devices are used, and the ML model is partitioned across them. Communication is much more expensive in edge clusters than in clusters within the cloud. Therefore, when partitioning an ML model between mobile and edge devices, there is a trade-off between computational capabilities and communication. At the same time, when parallelizing a model exclusively between edge and mobile devices, input data to the ML model can be kept on the device where the data was collected. This partitioning of input data can ensure the privacy of collected data as data is not transmitted to different devices.

Data parallelism [1] is partitioning across the input data dimension, where copies of the entire neural network are placed on multiple devices. Each device then processes subsets of the input data. Federated learning (FL) implements data parallelism in the training process while protecting private data by keeping input data on the edge and mobile devices on which the data is collected. Beyond standard data parallelism, the critical concept of FL revolves around a trade-off between communication efficiency and model accuracy. In FL devices can communicate continuously or at reduced intervals. Continuous communication keeps the model up-to-date. By contrast, more infrequent communication reduces the rate of convergence or, for the same amount of training, reduces accuracy due to outdated or incomplete data. To balance these factors, FL optimizes communication frequency to maintain accuracy, minimize overhead, and preserve data privacy [2].

In many mobile and edge scenarios, an entire ML model cannot fit on one device, and model parallelism is implemented, where the neural network is partitioned into sub-models, with each part of a model being placed on a separate device. This allows different parts of a model to be processed in parallel during either inference or training but requires communication of intermediate and internal data at the interface between partitions. Model and data parallelism can be combined, creating an ample design space for coarse-grain distributed ML.

The rest of this survey is structured as follows: Section 2 will focus on typical layer-based ML model architectures along with model and data parallelism and their respective partitioning schemes. Section 3 in turn will explore different layer types in ML models and how those layers are grouped and partitioned. Section 4 will focus on challenges and future directions related to partitioning schemes. Finally, we will conclude in Section 5.

Chapter 2: Data and Model Partitioning

This chapter aims to show how a typical ML model architecture can be partitioned to exploit data and model parallelism at the general model architecture level, leaving finer-grained partitioning methods to be discussed in the following chapter.

Figure 2.1 illustrates how a simple model can be partitioned within the data (d), model path/branch (m), and layer (l) dimensions. Assuming there are i pieces of input data, we can partition across the data (d) dimension by sending a specific amount of data samples to one device and the rest of the data samples to a similar model on another device. Neural network models are typically organized as a sequence of layers $L_X ... L_Y$, where multiple layers exist in ML networks, with branches between layers, spanning the model partitioning design space. For example, a partitioning configuration can determine that layers $L_{00},\ L_{01},\ L_{02},\ L_{03}$ can be executed on one device while layers L_{10} and L_{11} are executed on another device in parallel. The output of both sets of layers is then sent to L_Y. Even in this simple model illustrating model parallelism, the choice of partitioning methods is non-trivial and offers a diverse design space. For instance, instead of co-locating layers $L_{00},\ L_{01},\ L_{02},\ L_{03}$ on a single device, each layer could be allocated to a separate device. Moreover, the size of the design space increases as partitioning schemes differ depending on whether the model is being used for inference or training. With inference, devices may not have to communicate if the data is local. Whereas for training, communication must happen when gradients are updated; moreover, as in FL, the frequency of gradient updates must also be decided.

Table 2.1 surveys existing approaches exploring various forms of model and data parallelism. We categorize works based on training or inference contexts, cloud or edge computing environments, partitioning dimensions, computational benefits, communication requirements, memory advantages, and privacy considerations.

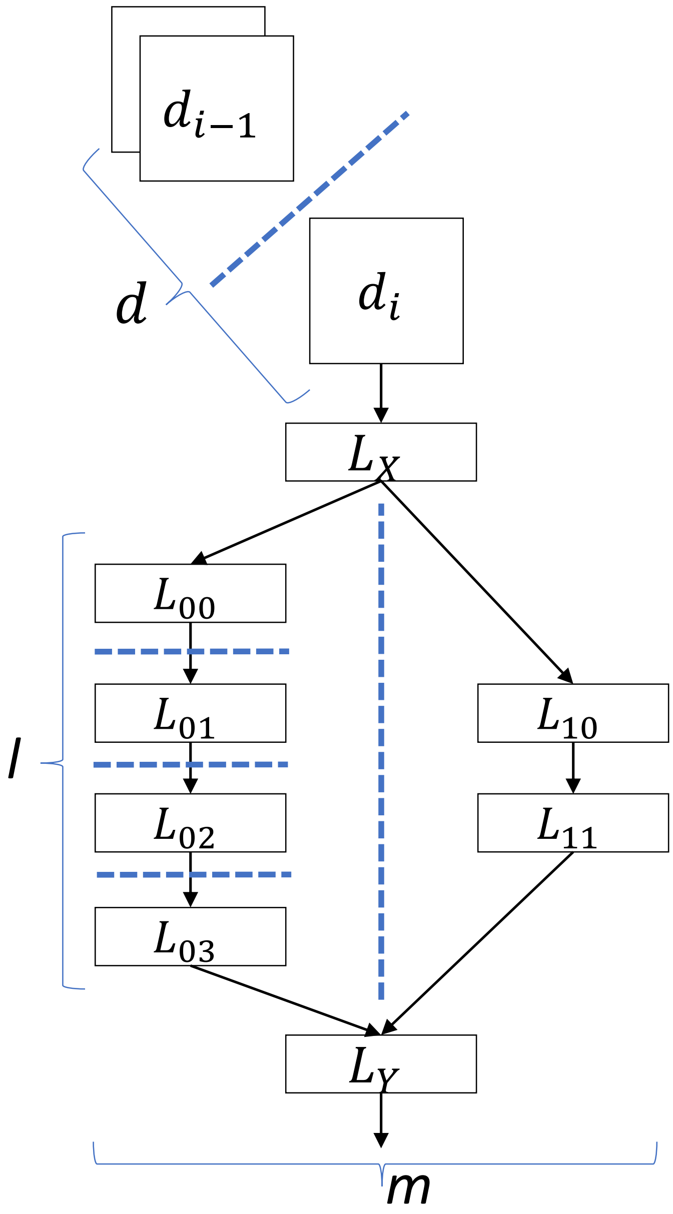

Figure 2.1: Data and model parallelism in a neural network.

Table 2.1: Model and data parallelism in ML networks.

	Inf./ Train.	Cloud/ Edge	part. dim.	comp. benefit	comm. req.	memory benefit	privacy
Cloud Training [3, 4, 5]	Train.	Cloud	$d/l/m$	throughput	weights/ data	weights	-
Parameter Server [6, 7]	Train.	Cloud/ Edge	$d/l/m$	latency	weights	weights/ data	-
Cloud-Assisted Inference [8, 9]	Inf.	Cloud/ Edge	$d/l/m$	throughput	weights/ data	weights	x
Federated Learning [10, 11, 12, 13, 14]	Train.	Cloud/ Edge	d	throughput	weights	-	x
Edge Inference [15, 16, 17, 18, 19, 20, 21]	Inf.	Edge	$d/l/m$	throughput	weights/ data	weights	x

Traditionally, powerful machines such as GPUs are used to train complex models in cloud computing by partitioning the training data batches onto different GPUs in a cluster in a data-parallel fashion, i.e., partitioning in the d dimension [22]. This method usually involves copies of the entire model to be placed on different machines, with each worker aggregating its gradients during training until model convergence is achieved [5]. More recently, finer ML model partitioning has been implemented in cloud contexts to train ML models. Partitioning along the l and m dimensions allow parts of the model to be offloaded to devices that can handle the task's computational complexity and increase the system's throughput [3]. More recently, works such as [4] have increased the search space to partitioning in the d, l, and m as well as intra-layer partitioning, introducing algorithms to determine the most efficient partitioning scheme based on each device in the cloud cluster's computational capabilities and the communication latency of the system.

To further decrease communication overheads and increase parallelism, synchronization requirements are relaxed in the parameter server model during cloud training. For example, the work in [6] introduces asynchronous communication between worker nodes and a server. In this case, the worker nodes collect data and process parts of the ML model. Simultaneously, the server tracks globally shared parameters independently, which helps decrease the system's latency by enabling concurrent execution and avoiding the need for constant synchronization. However, determining when to update the model parameters and distribute these parameters back to worker nodes is a non-trivial problem as parameters may have dependencies and different convergence rates, further increasing the complexity of the design space for training on cloud and edge devices [7].

As the popularity of edge computing has increased, many works are leveraging the power of the cloud and edge devices to jointly perform inference tasks. In addition to addressing resource constraints on the edge, the system's throughput increases as the inference process is pipelined between edge devices and the cloud. Since input data is stored on the device it was collected on, with only model features being sent to the cloud for processing, privacy can be protected with cloud-assisted inference [8]. However, communication time remains an issue as input and output data must be communicated between devices and the cloud. To mitigate the communication costs incurred by offloading computation to the cloud works such as [9] further partition the ML network based on each device's computational capabilities to increase the system's throughput.

FL is a machine learning approach where a centralized model, usually located in the cloud, is trained collaboratively across decentralized edge devices, allowing for privacy-preserving and efficient model training without centralized data aggregation. Each edge device holds a local model that is then trained on the input data received on each device. Finally, after a specified period, each edge device sends its updated model weights to the cloud to be aggregated. After this aggregation occurs, the updated weights are sent back to each edge device in the cluster, and another iteration begins [10]. Privacy is maintained as data is kept on the device on which it was collected, and this input data is not sent to the central cloud server. In addition to privacy, FL offers the benefit of increased system throughput due to devices working in parallel. Although this partitioning approach has its advantages, it may overlook the fairness of data distribution across the device cluster, resulting in statistical heterogeneity that could increase the convergence time of the model [13, 12]. However, communication costs are significant as weight updates must be communicated between the cloud server and each edge device. Works such as [11] and [14] aim to increase the convergence rate and increase throughput by determining when to perform the global parameter aggregation as well as taking into account the heterogeneity of the system.

Finally, pure edge inference aims to keep all inference tasks on edge devices. This ensures full data privacy. This partitioning scheme typically keeps input data on the device it is collected on while partitioning the model across the edge cluster along the layer-wise or per-branch dimensions based on each device's computational and memory constraints [21, 16, 15]. Moreover, approaches in this space also account for the fact that some devices may be idle while other devices have a large workload; therefore, tasks on a single device can exploit the idle computational power of other devices in the network [18]. Additionally, in this space, some works focus on memory benefits during partitioning [19], while other works focus on system throughput and data transmission [20, 17].

In summary, data and model parallelism provide practical strategies for scaling DL tasks while optimizing computational resources. A vast amount of literature in the field details various partitioning schemes for data and model parallelism with different advantages and disadvantages. However, achieving finer control and customization in resource allocation within ML networks requires partitioning at the individual layer level. This approach enables tailored optimization to accommodate diverse computational demands and constraints across DL environments and will be discussed in the next chapter.

Chapter 3: Layer Partitioning

We will continue exploring how to partition ML models within individual layers. However, each layer type may have different partitioning dimensions. This means multiple parallelism techniques can be used, depending on the specific layer type.

3.1 Fully Connected Layers

A fully connected (FC) layer and its corresponding weights in a neural network can be represented as matrix-vector multiplications. In Figure 3.1, we note the d, m, and n dimensions. The d dimension represents the input samples to the network, where each sample is a vector with dimension m. This input data vector is then multiplied with a weight matrix of dimensions $m \times n$, representing n neurons with m weighted inputs, producing a sequence of d output vectors of dimension m*'*, which is equivalent to n. In addition to exploiting data parallelism by partitioning along the d dimension as described in Chapter 2, FC layers provide additional intra-layer parallelizing opportunities by partitioning along the m or n dimensions.

Table 3.1 summarizes the different approaches for partitioning FC layers. Again, we categorize approaches based on inference and/or training, cloud and/or edge deployment, partitioning dimensions, computational benefits, communication requirements, memory benefits, and whether the approach protects private data.

The works in [15, 23, 24] focus on partitioning the inputs and outputs of FC layers through layer output and input partitioning. In layer output partitioning (LOP), the input vector of size m is multiplied by a subset of neurons, evenly distributed among a chosen subset size of n devices. After the summation of each subset is computed, an activation function is applied to each subset. Finally, each activated output is sent to the same device and concatenated to create the full output vectors

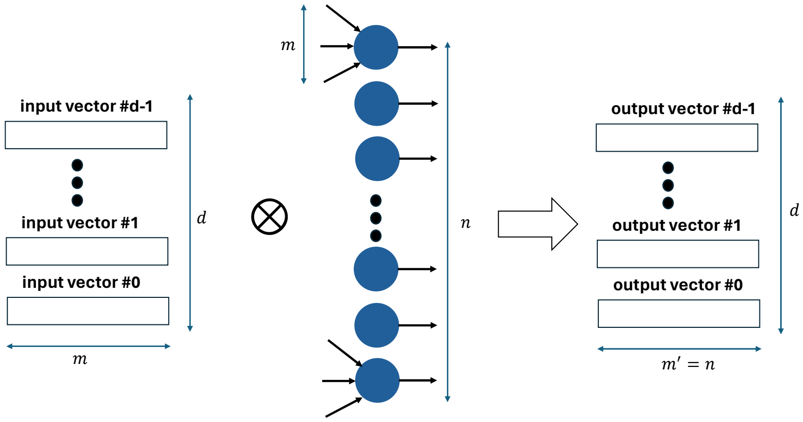

Figure 3.1: Fully connected layer parallelism.

of length n. As a result of LOP, each device has an even memory footprint and communication cost related to the size of each subset n. In layer input partitioning (LIP), m is split into subsets and placed on separate devices. Then, each subset is multiplied by n weights and sent to a final device that calculates the summation of all of the subsets and applies the activation function to each output vector of length n. LOP can outperform LIP because LOP communicates less data overall as more values are set to zero when the activation function is applied before communication. Additionally, both methods decrease the total memory required on each device and increase the system's throughput as the inference tasks can be pipelined. However, this method does not protect data privacy, as all data is transmitted to the device that holds the network's input layer.

LIP and LOP support the fusing of operations, where the partitioned outputs of an FC layer on which LOP has been applied can be directly fed as partitioned inputs to a subsequent LIP setup without having to assemble complete intermediate

Table 3.1: Partitioning schemes for fully connected layers.

	Inf./ train.	Cloud/ Edge	part dim.	comp. benefit	comm. req.	memory benefit	privacy
Output-Based Partitioning [15, 23, 24]	Inf.	Edge	j	throughput	output	output/ weights	-
Input-Based Partitioning [15, 23]	Inf.	Edge	i	throughput	input	input/ weights	-
Hybrid [25]	Train.	Edge	i/j	throughput	weights	input/ output/ weights	x

vectors, i.e., without the need to communicate between devices. Note that in the case of FC layers, such a fused LOP-LIP combination can, at maximum, encompass two layers.

In addition to input and output partitioning for inference, works such as [25] describe an approach for FC layer training that uses ideas from both federated learning and distributed training by partitioning fully connected layers across the m and n dimensions with the same number of layers as the original model on edge devices. The sub-models are then trained, and weight updates are shared, similar to FL, which in turn decreases the synchronization overhead as synchronization only needs to happen once after the sub-models are trained. This increases throughput in the system and allows networks to be fully trained on edge devices with limited memory. This approach protects private data as all input data is kept on the collected device.

While fully connected layers play an essential role in ML models, computer vision tasks have gained popularity, making convolutional layers ubiquitous. This leads to more challenges when determining how to run these computer vision networks on memory-constrained devices.

3.2 Convolutional Layers

Figure 3.2 describes the architecture of convolutional layers. Multiple filters *f*, denoted with height and width s and depth c, are convolved with the input tensors to that layer, of width w, height h, and channel c dimensions, forming a so-called feature map of size $w \times h$ and depth c. This produces an output tensor (feature map) of width w', height h', and depth c'. The size of the output feature map depends on the input width and height $(w \times h)$, padding, and striding of how the filters are applied, while each filter produces one output channel, i.e., the number of output channels c' is equal to the number of filters (f).

Table 3.2 summarizes partitioning strategies for convolutional layers. In general, we can distinguish strategies based on their focus on the partitioning of feature maps or filter weights.

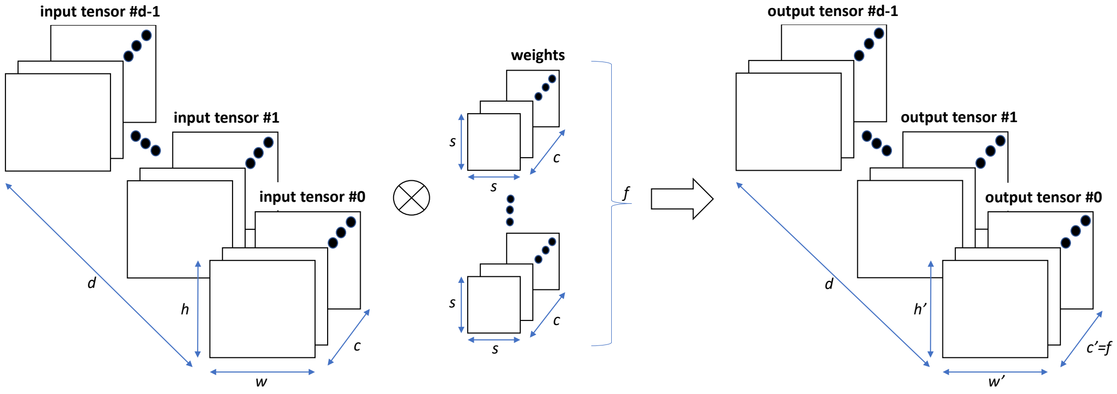

Figure 3.2: Convolutional layer parallelism.

3.2.1 Feature Map Partitioning

A common strategy for partitioning convolutional layers is to tile the layer across the h and w dimensions. This exploits the inherent locality in convolutions, where each device processes one input tile to produce a corresponding output tile. MoDNN [24] partitions convolutional layers in the h or w dimensions to minimize

Table 3.2: Partitioning schemes for convolutional neural networks.

	Inf. /Train.	Cloud /Edge	part. dim.	comp. benefit	comm. req.	memory benefit	privacy
Feature Partitioned Inference [24, 26, 27, 28, 29]	Inf.	Edge	w/h	latency	input/ output	input/ output	x
Weight Partitioned Inference [30, 31]	Inf.	Edge	c/f	latency	weights	weights	x
Weight Partitioned Training [32, 33]	Train.	Cloud	c/f	latency	weights	weights	-

the need for nodes in the cluster to communicate. While this reduces data dependencies and the memory required to store intermediate feature map data, it maintains layer-by-layer execution, potentially causing network bottlenecks and lacking dynamic adaptation to varying computing demands.

Layer fusion, introduced in [34], aims to further reduce data transmission in a network. In layer fusion, the outputs of one layer of the network are sent directly as the inputs to the next layer of the network on the same device, bypassing the need to communicate intermediate feature map data between devices. However, since data regions overlap in convolutional operations, as shown in Figure 3.3, the overlapping segments of the nodes must still be communicated. Consequently, device dependency increases, resulting in the reliance on communication from other devices for shared data. However, in layer fusion, data privacy is protected as most information is kept local on each edge device and not offloaded to other devices or the cloud for further processing.

DeepThings [26] implements a fusing approach along with tile-based partitioning by dividing convolutional layers into independent tasks based on local regions for parallel execution. This approach reduces memory usage and communication overhead by fusing intermediate feature maps within edge nodes, leading to more efficient data transmission than MoDNN. In addition to the previously discussed works, a large amount of research has been done on the topic, such as introducing heterogeneous devices into the edge cluster [27, 29, 28] to maximize resource utilization and decrease the total latency of the system.

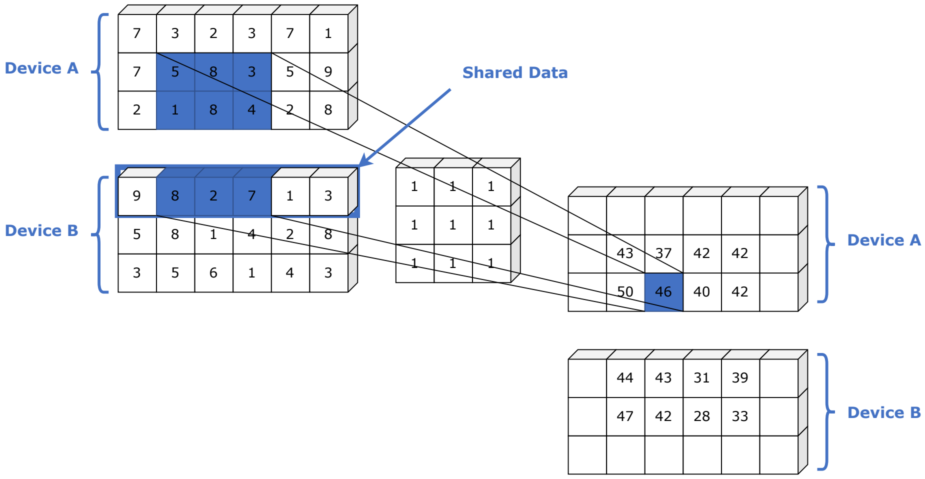

Figure 3.3: Shared data in distributed convolutional operations.

3.2.2 Channel, Filter, and Weight Partitioning

In convolutional neural networks (CNNs), the channel dimension (c) is critical for achieving high accuracy especially in later layers of deep CNNs.[35]. Therefore, network computational complexity increases as the c dimension increases. As a result, partitioning convolutional layers in the channel dimension c is a popular method that decreases the latency of a model and increases throughput. Partitioning in the c dimensions splits both feature maps and filters to process a subset of channels on each device. This reduces memory requirements for both feature map and weight data, but requires communication and summation of the outputs produced by each partition/device to obtain output tensors.

Alternatively, filter partitioning, or partitioning in the f dimension simply assigns one complete filter to each partition/device, where each partition/device produces one channel of the output feature map, where channels only need to be assembled to form the complete map. This partitioning strategy allows for effective distribution of computation in weight- or filter-dominated layers across devices while minimizing communication and memory overhead, optimizing the execution of CNNs on resource-constrained systems.

Partitioning a network in both the c and f dimensions aims to further reduce the overhead incurred by partitioning in the h and w dimensions alone, reducing memory and communication overhead while enhancing efficiency [30, 31]. This approach considers resource constraints in edge clusters and demonstrates performance improvements in well-known CNNs. However, note that in contrast to feature map partitioning, similar to FC layer input-output partitioning, this work only allows for pairwise fusing of filter partitioned with channel partitioned layers, i.e. weight partitioning does not directly support arbitrary fusing of layers, therefore, compared to feature partitioning techniques, this partitioning scheme has a larger communication overhead.

The work in [32] proposes Xception, a CNN architecture based on depthwise separable convolutions, which inherently allow for partitioning of depthwise and pointwise convolution operations across the channel (c) and filter (f) dimensions, respectively. This approach enhances parameter efficiency and model performance in image classification tasks while reducing computational complexity, facilitating faster training. By employing depthwise separable convolutions, Xception achieves a smaller model size due to fewer parameters, demonstrating effective channel partitioning strategies to optimize CNN efficiency. However, Xception does not decrease the communication overhead between devices in a cluster. In contrast, the work in [33] emphasizes channel and filter parallelism to accelerate large-scale CNN training. This approach enables strong scaling, reduces communication overhead, and improves memory efficiency by distributing computation across channels and filters. While it introduces additional communication during training, volume and memory usage is optimized, further highlighting the importance of channel partitioning for enhancing CNN scalability and efficiency in training scenarios.

3.3 Recurrent Layers

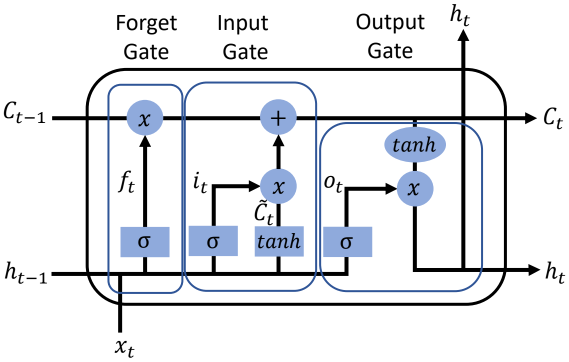

Figure 3.4: Long Short-Term Memory (LSTM) cell.

Recurrent Neural Networks (RNNs) represent a class of artificial neural networks designed explicitly for processing sequential data. Their architecture incorporates recurrent connections, enabling information to propagate across time steps. This recurrent structure can be conceptualized as an unrolled network, where each iteration receives input from the current element in the sequence and the hidden state of the previous iteration. This hidden state is a memory mechanism encoding information from past inputs and influencing the network's response to subsequent elements. However, RNNs are susceptible to the vanishing gradient problem, where gradients diminish as they backpropagate through time, hindering the network's ability to learn long-range dependencies within the sequence.

To address the limitations of RNNs, precisely the vanishing gradient problem, Long Short-Term Memory Networks (LSTMs) were introduced as shown in Figure 3.4. LSTMs, a specialized variant of RNNs, augment the architecture with a gated cell state mechanism. This cell state acts as a ”long-term memory,” allowing the network to retain information over extended periods. Three gates regulate the flow of information into and out of the cell state: the forget gate, which selectively discards irrelevant information; the input gate, which determines what new information to store; and the output gate, which controls the information retrieved from the cell state for generating the current output. The necessary equations used in an LSTM network are shown below, where the W variables form matrices that represent the network's weights.

$$i_t = \sigma(x_t U^i + h_{t-1} W^i) \tag{3.1}$$

$$f_t = \sigma(x_t U^f + h_{t-1} W^f) \tag{3.2}$$

$$o_t = \sigma(x_t U^o + h_{t-1} W^o) \tag{3.3}$$

$$\tilde{C}_t = tanh(x_t U^g + h_{t-1} W^g) \tag{3.4}$$

$$C_t = \sigma(f_t * C_{t-1} + i_t * \tilde{C}_t) \tag{3.5}$$

$$h_t = tanh(C_t) * o_t \tag{3.6}$$

By mitigating the vanishing gradient problem, LSTMs excel at capturing long-range dependencies in sequential data, making them superior to traditional RNNs for tasks requiring extended memory, such as natural language processing and time series prediction [36].

To optimize performance and scalability, partitioning LSTM layers in ML involves splitting the LSTM computations across different dimensions, such as the forget, input, and output gates. Table 3.3 details partitioning methods of recurrent layers.

Table 3.3: Existing works on RNN partitioning.

	Inf./ Train.	Cloud/ Edge	part. dim.	privacy
Gate Partitioned [37, 38]	Inf./ Train.	Edge	σ/tanh	x
Weight Partitioned [39, 40, 41]	Inf./Train.	Edge	W	x
Model Partitioned [39]	Train.	Cloud	$W/x/h$	

3.3.1 Gate-Based Partitioning

One effective way to partition LSTM layers is through gate-wise decomposition. Each LSTM gate (forget, input, and output) can be treated as an independent computational unit. This approach allows for parallelization and distribution of computations. For instance, the LSTM layer can be divided into sub-layers corresponding to each gate. In this setup, the forget gate sub-layer handles computations related to forgetting information from the previous cell state, and the input gate sub-layer manages input modulation and decides what new information to store. The output gate sub-layer controls the output generation. By partitioning in this manner, each sub-layer can operate independently and efficiently utilizing hardware resources in distributed systems. Additionally, parameter sharing across these sub-layers can be leveraged to reduce memory footprint and increase training speed. This partitioning strategy optimally distributes the workload, enhancing the scalability and performance of LSTM networks in ML tasks.

Gate partitioning has been applied to custom hardware implementation of FSMs on FPGAs [37]. Due to the recurrent nature of LSTMs, traditional hardware does not allow for maximum performance. CPUs do not offer large amounts of parallelism, and small RNNs do not fully benefit from the parallelization of GPUs.

Therefore, a case is made for specialized hardware to run inference tasks on these models. To increase parallelization in computation tasks, two sigmoid and one tanh gates are implemented to allow equations used in the LSTM network to occur in parallel if no dependencies exist. The work in [38] verifies that the FPGA approach is more efficient than CPU and GPU-based approaches due to the ability to extract fine-grained parallelism in LSTM modules.

3.3.2 Weight-Based Partitioning

The weight-based partitioning methods during inference focus on achieving scalable RNN acceleration on FPGA platforms. The work in [41] introduces three levels of parallelism—matrix-level, operation-level, and layer-level—to optimize RNN processing across multiple FPGAs. Matrix-level parallelism (weight-based partitioning) simply takes matrix-vector multiplications, where the matrix is the weight matrix, W, and partitions these multiplications into independent execution units based on either the rows or columns of W. Operation-level parallelism takes entire matrix-vector multiplications in a layer and executes them on different functional units. For example, the multiplications involving x_t and h_t would be placed on different functional units and executed in parallel. Layer-level parallelism is placing each layer in the network on separate FPGAs. These parallelism approaches also include analyzing dependencies within RNNs and implementing software pipelining to enhance hardware utilization. Additionally, the work in [40] introduces Bank-Balanced Sparsity (BBS) to achieve high accuracy and hardware efficiency by partitioning weight matrix rows into equally sized banks and applying fine-grained pruning. Moreover, the work in [39] tackles the challenge of training large LSTM networks by proposing Factorized LSTM (F-LSTM), which decomposes the LSTM weight matrix (W) into the product of two smaller matrices (W_1 and W_2), reducing the total parameter count and computational complexity. Together, these works showcase diverse weight partitioning strategies tailored for RNN acceleration on FPGA platforms, emphasizing parallelism exploitation, hardware awareness, and software-hardware co-design to optimize performance based on specific model requirements and hardware constraints.

3.3.3 Model Partitioning

Layer-based partitioning in [41] described in Section 3.3.2 realizes a form of model parallelism. In addition, the work in [39] also proposes an alternative model partitioning scheme for training LSTMs called Group LSTM (G-LSTM). This scheme partitions the LSTM cell, inputs (x_t), and hidden states (h_t) into independent groups, each operating on a subset of features with its own set of parameters. G-LSTM enables parallel processing during training and reduces the overall parameter count. The G-LSTM model can be interpreted as an ensemble of smaller LSTM models operating on different feature subsets concatenated to preserve feature independence. Both G-LSTM and F-LSTM (discussed in Section 3.3.2) in [39] significantly reduce parameter counts and training times while maintaining accuracy, facilitating the training of more extensive LSTM networks on constrained hardware resources for improved model complexity and performance exploration.

Chapter 4: Challenges and Future Directions

Distributed learning holds immense promise for scaling and optimizing machine learning applications; however, several critical challenges must be addressed to realize its full potential and ensure its effective deployment in practical settings. One primary challenge is communication overhead, particularly in resource-constrained edge and mobile environments. Frequently, data exchange between distributed devices during training or inference can lead to bottlenecks and latency issues. Optimizing communication protocols, implementing efficient data compression techniques, and developing effective model partitioning strategies are crucial to mitigate this challenge.

Another significant hurdle is system heterogeneity within distributed learning environments. These systems often involve diverse devices with varying computational capabilities, memory capacities, and communication bandwidths. Addressing this heterogeneity requires the development of adaptive and robust algorithms capable of efficiently distributing workloads and gracefully handling device failures or communication disruptions. Moreover, ensuring data privacy and security remains paramount, particularly in scenarios like fully distributed ML, where data privacy is a top concern as potentially private data needs to be communicated between devices.

Efficient resource management and scheduling across distributed devices present additional complexities. This involves optimizing the allocation of computational resources, considering factors like device availability, energy consumption, communication costs, and task deadlines. Developing intelligent resource management and scheduling algorithms is crucial for optimizing system performance and reliability.

As ML models become increasingly complex, model optimization and compression emerge as essential strategies to ensure efficient distributed execution. Techniques such as model pruning, quantization, and knowledge distillation enable the reduction of model size and computational requirements while compromising accuracy. Combining these methods with the current distributed ML techniques described in this report would allow for faster inference times due to increased throughput and lower latency.

Looking ahead, several promising future directions can further advance distributed learning. Developing adaptive and dynamic partitioning strategies that respond to changing system conditions, leveraging specialized hardware accelerators for specific tasks or layer types, and fostering standardization and interoperability through established protocols and frameworks are critical areas for future exploration and innovation.

By systematically addressing these challenges and actively pursuing these promising avenues, distributed learning will continue to evolve and play a transformative role in shaping the future landscape of machine learning applications.

Chapter 5: Summary and Conclusions

This survey has explored the multifaceted landscape of distributed learning, highlighting its potential to address the growing computational demands of modern machine learning models. We have examined the core principles of data and model parallelism, analyzing how models are strategically partitioned across diverse devices and environments to optimize resource utilization and performance. We have shed light on the critical trade-offs between computational efficiency, communication overhead, and memory constraints by delving into various partitioning schemes for different layer types, including fully connected, convolutional, and recurrent layers.

Through the lens of distributed learning, we have gained valuable insights into optimizing model training and inference processes while safeguarding data security. The challenges and future directions discussed underscore the need for continued research and development in communication optimization, handling system heterogeneity, and protecting private data.

Traditionally, distributed learning strategies have been applied in different contexts, such as on the edge or within the cloud. However, proposed solutions to partitioning these networks share the same design space and contain many similarities. Therefore, distributed learning can unlock new possibilities for efficient, scalable, and privacy-aware machine learning applications across various domains by fostering collaboration and innovation within this field.

Acknowledgments

We would like to thank Kamyar Mirzazad Barijough for providing some of the initial figures and ideas for this report.

Bibliography

[1] P. Joshi, M. Hasanuzzaman, C. Thapa, H. Afli, and T. Scully, "Enabling all in-edge deep learning: A literature review," *IEEE Access*, vol. 11, pp. 3431–3460, 2023.

[2] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[4] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[5] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia. (2021) Efficient large-scale language model training on GPU clusters using megatron-LM. [Online]. Available: http://arxiv.org/abs/2104.04473

[6] M. Li, "Scaling distributed machine learning with the parameter server," in *Proceedings of the International Conference on Big Data Science and Computing*, 2014.

[7] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "STRADS: A distributed framework for scheduled model parallel machine learning," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[8] S. Teerapittayanon, B. McDanel, and H. Kung, “Distributed deep neural networks over the cloud, the edge and end devices,” in *International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 328–339.

[9] T. Mohammed, C. Joe-Wong, R. Babbar, and M. D. Francesco, “Distributed inference acceleration with adaptive dnn partitioning and offloading,” in *IEEE Conference on Computer Communications (INFOCOM)*, 2020, pp. 854–863.

[10] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, vol. 54, 2017, pp. 1273–1282.

[11] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, “When edge meets learning: Adaptive control for resource-constrained distributed machine learning,” in *IEEE Conference on Computer Communications (INFOCOM)*, 2018, pp. 63–71.

[12] H. B. McMahan *et al.*, “Advances and open problems in federated learning,” *Foundations and Trends in Machine Learning*, vol. 14, no. 1, 2021.

[13] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions,” *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[14] Z. Wang, H. Xu, Y. Xu, Z. Jiang, and J. Liu, “CoopFL: Accelerating federated learning with DNN partitioning and offloading in heterogeneous edge computing,” *Comput. Netw.*, vol. 220, no. C, Jan. 2023.

[15] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, “Toward collaborative inferencing of deep neural networks on internet-of-things devices,” *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.

[16] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, “Adaptive parallel execution of deep neural networks on heterogeneous edge devices,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 195–208.

[17] F. Xue, W. Fang, W. Xu, Q. Wang, X. Ma, and Y. Ding, “EdgeLD: Locally distributed deep learning inference on edge device clusters,” in *International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 613–619.

[18] Z. Huai, B. Ding, H. Wang, M. Geng, and L. Zhang, “Towards deep learning on resource-constrained robots: A crowdsourcing approach with model partition,” in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2019, pp. 989–994.

[19] E. Tang and T. Stefanov, “Low-memory and high-performance CNN inference on distributed systems at the edge,” in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC)*, 2021.

[20] J. Zhou, Y. Wang, K. Ota, and M. Dong, “AAIoT: Accelerating artificial intelligence in IoT systems,” *IEEE Wireless Communications Letters*, vol. 8, no. 3, pp. 825–828, 2019.

[21] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, “Distributed perception by collaborative robots,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3709–3716, 2018.

[22] X.-W. Chen and X. Lin, “Big data deep learning: Challenges and perspectives,” *IEEE Access*, vol. 2, pp. 514–525, 2014.

[23] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Fully distributed deep learning inference on resource-constrained edge devices," in *nternational Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2019, pp. 77–90.

[24] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 1396–1401.

[25] B. Yuan, C. R. Wolfe, C. Dun, Y. Tang, A. Kyrillidis, and C. Jermaine, "Distributed learning of fully connected neural networks using independent subnet training," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, 2022.

[26] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[27] L. Zeng, X. Chen, Xu Chen, Xu Chen, Z. Zhou, L. Yang, J. Zhang, and J. Zhang, "CoEdge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[28] Z. Gao, S. Sun, Y. Zhang, Z. Mo, and C. Zhao, "EdgeSP: Scalable multi-device parallel dnn inference on heterogeneous edge clusters," in *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2021, p. 317–333.

[29] X. Wang, Z. Yang, J. Wu, Y. Zhao, and Z. Zhou, "EdgeDuet: Tiling small object detection for edge assisted autonomous mobile vision," in *IEEE Conference on Computer Communications (INFOCOM)*, 2021, pp. 1–10.

[30] S. Dey, A. Mukherjee, A. Pal, and P. Balamuralidhar, "Partitioning of CNN models for execution on fog devices," in *Proceedings of the 1st ACM International Workshop on Smart Cities and Fog Computing*, Shenzhen China, Nov. 2018, pp. 19–24.

[31] R. Stahl, A. Hoffman, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "DeeperThings: Fully distributed CNN inference on resource-constrained edge devices," *International Journal of Parallel Programming*, vol. 49, no. 4, pp. 600–624, 2021.

[32] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1251–1258.

[33] N. Dryden, N. Maruyama, T. Moon, T. Benson, M. Snir, and B. Van Essen, "Channel and filter parallelism for large-scale CNN training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver Colorado, Nov. 2019, pp. 1–20.

[34] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, no. arXiv:1409.1556. arXiv, Apr. 2015.

[36] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[37] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *arXiv preprint arXiv:1511.05552*, 2015.

[38] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," in *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[39] O. Kuchaiev and B. Ginsburg, "Factorization tricks for LSTM networks," 2018, number: arXiv:1703.10722. [Online]. Available: http://arxiv.org/abs/1703.10722

[40] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, New York, NY, 2019, p. 63–72.

[41] D. Kwon, S. Hur, H. Jang, E. Nurvitadhi, and J. Kim, "Scalable multi-FPGA acceleration for large RNNs with full parallelism levels," in *ACM/IEEE Design Automation Conference (DAC)*, 2020.