Low-memory and High-performance CNN Inference on Distributed Systems at the Edge

Erqian Tang
LIACS, Leiden University
Leiden, the Netherlands
e.tang@liacs.leidenuniv.nl

Todor Stefanov
LIACS, Leiden University
Leiden, the Netherlands
t.p.stefanov@liacs.leidenuniv.nl

ABSTRACT

Nowadays, some applications need CNN inference on resource-constrained edge devices that may have very limited memory and computation capacity to fit a large CNN model. In such application scenarios, to deploy a large CNN model and perform inference on a single edge device is not feasible. A possible solution approach is to deploy a large CNN model on a (fully) distributed system at the edge and take advantage of all available edge devices to cooperatively perform the CNN inference. We have observed that existing methodologies, utilizing different partitioning strategies to deploy a CNN model and perform inference at the edge on a distributed system, have several disadvantages. Therefore, in this paper, we propose a novel partitioning strategy, called Vertical Partitioning Strategy, together with a novel methodology needed to utilize our partitioning strategy efficiently, for CNN model inference on a distributed system at the edge. We compare our experimental results on the YOLOv2 CNN model with results obtained by the existing three methodologies and show the advantages of our methodologies in terms of memory requirement per edge device and overall system performance. Moreover, our experimental results on other representative CNN models show that our novel methodology utilizing our novel partitioning strategy is able to deliver CNN inference with very reduced memory requirement per edge device and improved overall system performance at the same time.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Computing methodologies → Machine learning**.

KEYWORDS

Convolutional Neural Networks, Distributed System at the Edge, Low-memory, High-performance, CSDF

ACM Reference Format:
Erqian Tang and Todor Stefanov. 2021. Low-memory and High-performance CNN Inference on Distributed Systems at the Edge. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC '21) Companion (UCC '21 Companion), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3492323.3495629

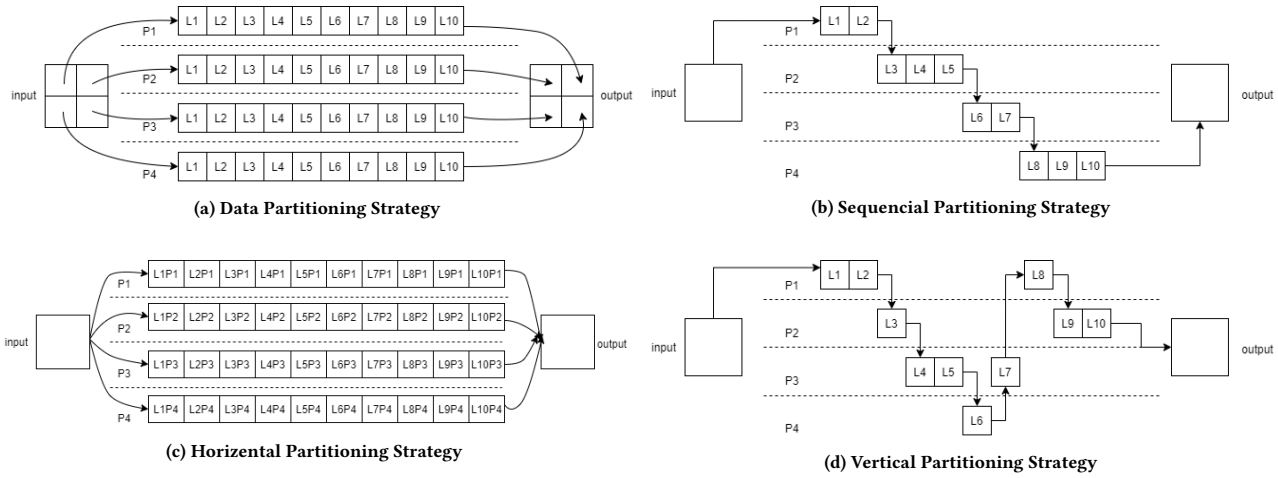
UCC '21 Companion, December 6–9, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-9163-4/21/12…$15.00
https://doi.org/10.1145/3492323.3495629

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have been intensively researched and widely used to perform various tasks in areas such as image recognition and natural language processing, due to their ability to process data in large scale and with high classification accuracy after training[3]. After a CNN model is trained, it can be deployed on different kinds of hardware to perform CNN inference on actual data within different applications. The CNN inference is usually computation-intensive and some CNN models are huge, i.e., they require a lot of memory to perform the inference. Powerful hardware like a server or a desktop usually has enough memory to fit a CNN model. However, some applications need CNN inference on resource-constrained edge devices that may have very limited memory capacity to fit a CNN model. For example: (1) In an Internet of Things (IoT) smart home application, edge devices like a light controller, or a temperature conditioner need to deploy and execute a natural language processing model in order to recognize person's input voices and generate output commands accordingly; (2) In a military mobile robots application, some tiny size robots like robotic ants and robotic insects need to invade into secret places to detect objects and activities, utilizing image recognition models. In such application scenarios, to deploy a CNN model and perform inference on a single edge device is not possible because the hardware platform has very limited resources in terms of memory and computation capacity to fit the CNN model. One approach to solve this issue is to perform CNN model compression (e.g. pruning, quantization, or knowledge distillation) [6, 8, 15] but such approach sacrifices the accuracy of the model to some extent. Another approach is to deploy only part of the CNN model on the edge device hardware platform and the rest of the CNN model on the cloud or remote server/desktop, but such approach may have data privacy issues like leakage of personal information or sensitive military information. Also, sending data from the edge device to the cloud or remote server/desktop and returning back results may lead to unacceptable CNN inference latency because of the internet communication and transmission distance. A third approach, which solves the aforementioned issues of the other two approaches, is to deploy the CNN model on a (fully) distributed system at the edge and take advantage of all available edge devices to cooperatively perform the CNN inference. The existing methodologies that use this approach to deploy a CNN model and perform inference at the edge on a distributed system can be divided in three main categories depending on the CNN model partitioning strategy used:

(1) Methodologies with Data Partitioning Strategy: The main features of the data partitioning strategy are shown in Row 3 of Table 1. The strategy may partition the input data given to every CNN layer (L_i) whereas the weights of every CNN layer are not



(a) Data Partitioning Strategy

(b) Sequencial Partitioning Strategy

(c) Horizental Partitioning Strategy

(d) Vertical Partitioning Strategy

Figure 1: Fully distributed CNN model partitioning strategies

CNN model partitioning strategy	items of a layer to partition		layers to include per partition	
	input data	weights	all	consecutive
Data Partitioning Strategy	Yes	No	Yes	Yes
Sequentially Partitioning Strategy	No	No	No	Yes
Horizontal Partitioning Strategy	No	Yes	Yes	Yes
Vertical Partitioning Strategy	No	No	No	No

Table 1: Features of CNN model partitioning strategies

partitioned. Each CNN model partition (P_i) includes all layers of the CNN model, where the layers operate on only part of their input data because the input data to a layer is partitioned. Different CNN model partitions have dependencies because some parts of the input data to layer L_i of partition P_i may have to be shared with layers from other partition. Figure 1(a) shows a CNN model with ten layers ($L_1 - L_{10}$) that is partitioned into four partitions ($P_1 - P_4$). The memory, required to deploy each partition, can be significantly reduced because the memory needed for storing the data that is exchanged between the layers of a partition can be significantly reduced when the number of partitions is large. Moreover, the different partitions of the CNN model have very little data dependencies, because only very small boundary parts of the input data to a layer may have to be shared with other partitions. This can help to exploit data-level parallelism among different partitions, which in other words, helps to increase the system performance. However, this strategy does not reduce the memory needed for storing the weights of all the layers of a partition. If a CNN model has many layers and/or some layers require huge number of weights to be stored, then this strategy may easily fail to fit a CNN model partition in an edge device due to insufficient memory on the device. An example of a methodology utilizing this partitioning strategy to deploy a CNN model and perform inference on a distributed system at the edge can be found in [16].

(2) Methodologies with Sequential Partitioning Strategy: The main features of the sequential partitioning strategy are shown in Row 4 of Table 1. This strategy partitions the layers of a CNN model such that each partition (P_i) includes consecutive CNN layers. The input data, given to a CNN layer (L_i) and its weights are not partitioned. For example, Figure 1(b) shows a CNN model with ten layers ($L_1 - L_{10}$) that is partitioned into four partitions ($P_1 - P_4$). The memory, required to deploy each partition, can be significantly reduced because both the memory needed for storing the data that is exchanged between the layers of a partition and the memory needed for storing the weights of the layers of a partition can be significantly reduced when the number of partitions is large. Moreover, this strategy supports the execution of partitions in a pipeline fashion, which may help to exploit both data-level and task-level parallelism and to increase the system performance. However, this strategy may not be able to evenly distribute the required memory and the computation workload of a CNN model among different partitions because each partition must include only consecutive CNN layers. As a consequence, the unevenly distributed memory of a CNN model may create partitions that require an edge device with higher memory capacity, and the unevenly distributed workload may lead to a lower system performance because the partition with the heaviest workload will become the bottleneck. An example of a methodology utilizing this partitioning strategy to deploy a CNN model and supporting inference on a distributed system at the edge can be found in [14].

(3) Methodologies with Horizontal Partitioning Strategy: The main features of the horizontal partitioning strategy are shown in Row 5 of Table 1. This strategy partitions the weights of every CNN layer (L_i) whereas the input data given to a CNN layer is not partitioned. Each CNN model partition (P_i) includes all layers of the CNN model, where the layers operate with only part of their weights because the weights of a layer are partitioned. Therefore, every CNN layer (L_i) is divided in several parts (L_iP_j) and distributed onto several partitions. Different parts of the same CNN layer have to communicate and synchronize with each other because all the output data from part L_iP_j has to be concatenated with the output data from the other parts of the same layer. For example, Figure 1(c) shows a CNN model with ten layers ($L_1 - L_{10}$) that is partitioned into four partitions ($P_1 - P_4$). The memory, required to deploy each partition, can be significantly reduced because the memory needed for storing the weights of each partition can be significantly

reduced when the number of partitions is large. However, this strategy does not reduce the memory needed for storing the data that is exchanged between the layers of a partition. If a CNN model requires huge amount of data to be exchanged between layers, this strategy may easily fail to fit the CNN model partitions on edge devices. In addition, even though different partitions of the CNN model may exploit task-level parallelism, the aforementioned data communication and synchronization among the parts of a CNN layer may prevent any increase of the system performance. An example of a methodology utilizing this partitioning strategy to deploy a CNN model and perform inference on a distributed system at the edge can be found in [4, 10, 13].

In order to avoid the disadvantages of the aforementioned strategies, in this paper, we propose a novel partitioning strategy, called Vertical Partitioning Strategy, together with a novel methodology needed to utilize our partitioning strategy efficiently. Our novel partitioning strategy, introduced in Section 4.1, reduces both the memory needed to store weights and the memory needed to exchange data between layers of a partition. At the same time, our strategy enables even distribution of the required memory and computation workload of a CNN model among different partitions. In addition, our strategy helps to increase the overall system performance because it supports the execution of the partitions in a pipeline fashion which exploits both data-level and task-level parallelism. Our novel methodology, which enables the utilization of our partitioning strategy efficiently, consists of two main steps, introduced in Section 4. In Step 1, an efficient partitioning of a CNN model onto a fully distributed system of edge devices is obtained using a genetic algorithm (GA) to evenly distribute the required memory and computation workload of the CNN model onto the edge devices. In Step 2, based on the efficient partitioning obtained in Step 1, the CNN model is converted into a functionally equivalent Cyclo-Static Dataflow (CSDF) application model [5]. Unlike the CNN model, the CSDF model explicitly specifies task- and data-level parallelism, available in a CNN, as well as it explicitly specifies the tasks communication and synchronization mechanisms, suitable for efficient execution of the CNN on a fully distributed system at the edge. The experimental results, obtained by using our novel partitioning strategy and methodology on real-world CNNs implemented onto fully distributed systems that consist of 2 to 10 Nvidia Jetson-TX2 MPSoCs embedded platforms, are compared with results obtained from three existing methodologies that utilize the three aforementioned partitioning strategies illustrated in Figure 1(a), 1(b), and 1(c). This comparison shows that our proposed partitioning strategy and methodology are able to deliver CNN inference on a fully distributed system at the edge with: (1) a lower memory requirement per edge device and much higher system performance, compared with the methodology in [16] which utilizes the data partitioning strategy; (2) a lower memory requirement per edge device and higher system performance, compared with methodologies utilizing the sequential partitioning strategy, like in [14]; (3) a similar memory requirement per edge device and much higher system performance, compared with methodologies utilizing the horizontal partitioning strategy, like in [13].

The remainder of the paper is organized as follows: Section 2 gives an overview of the related work. Section 3 introduces the background material needed for understanding our methodology proposed in this paper. Section 4 presents in detail our proposed partitioning strategy and methodology. Section 5 provides the experimental results, and Section 6 ends the paper with conclusions.

2 RELATED WORK

The authors in [16] propose a novel methodology, called DeepThings, which utilizes only partly the data partitioning strategy introduced in Section 1. More specifically, the CNN layers with huge input/output data (e.g., early stage convolutional layers) are partitioned utilizing the data partitioning strategy and distributed on resource-constrained edge devices. The CNN layers with huge number of weights (e.g., fully connected layers) are not partitioned and they are deployed on one powerful gateway device. The experimental results on the YOLOv2 CNN model show that this methodology is able to: (1) reduce the memory needed per device by a maximum of 68% on a system with 2 to 7 edge devices, compared to a system with 1 device; (2) achieve overall CNN inference speedups of 1.3× to 1.6× on 2 to 7 edge devices compared to 1 edge device. However, this methodology will always require a powerful gateway device to deploy part of the CNN (e.g. 43.2% for YOLOv2) because some CNN partitions in this methodology are not evenly constructed in terms of memory requirements and computation workload. Moreover, the part of the CNN which is deployed on the gateway device becomes the bottleneck of the system because it is not partitioned and it limits the system performance. In contrast, our methodology supports even distribution of the CNN model in terms of both memory and computation workload. So, our methodology does not require a powerful gateway device to fit part of the CNN and the distributed CNN model does not have a bottleneck which limits the system performance. The experimental results on YOLOv2 show that our methodology is able to: (1) reduce the memory needed per device, by 48.2% to 74.3%, on 2 to 7 devices, compared to 1 device; (2) achieve overall CNN inference speedups of 1.9× to 4.7× on 2 to 7 edge devices, compared with 1 edge device.

Compared with [16], the methodology in [13] supports even distribution of a CNN model on resource-constrained edge devices and does not require a powerful gateway device. This methodology utilizes the horizontal partitioning strategy (introduced in Section 1) on the whole CNN. The experimental results on YOLOv2 show a proportional memory reduction rate (66.7% to 85.8%) per edge device with the increase of the number of the edge devices (3 to 7). However, in this methodology, every CNN layer is divided in several parts and distributed onto several partitions. Different parts of the same CNN layer have to communicate and synchronize with each other because all the output data from one part has to be concatenated with the output data from the other parts of the same layer which limits the system performance. The experimental results on YOLOv2 show that the system performance speedup is limited to only 1.2× to 1.6× on 3 to 7 edge devices, compared to 1 edge device. In contrast, in our methodology, different partitions have much less dependencies between each other, so complex synchronization and huge communication cost are avoided. As a result, our methodology can achieve much better system performance speedup, e.g., 2.6× to 4.7× on 3 to 7 edge devices for YOLOv2 inference. Moreover, our methodology can achieve a similar memory reduction rate (64.1%

to 74.3%) per edge device with the increase of the number of the edge devices (3 to 7), compared with [13].

3 BACKGROUND

In this section, we describe the Convolutional Neural Network (CNN) model and the Cyclo-Static Dataflow (CSDF) model that are essential for understanding our novel methodology, which efficiently utilizes our novel vertical partitioning strategy.

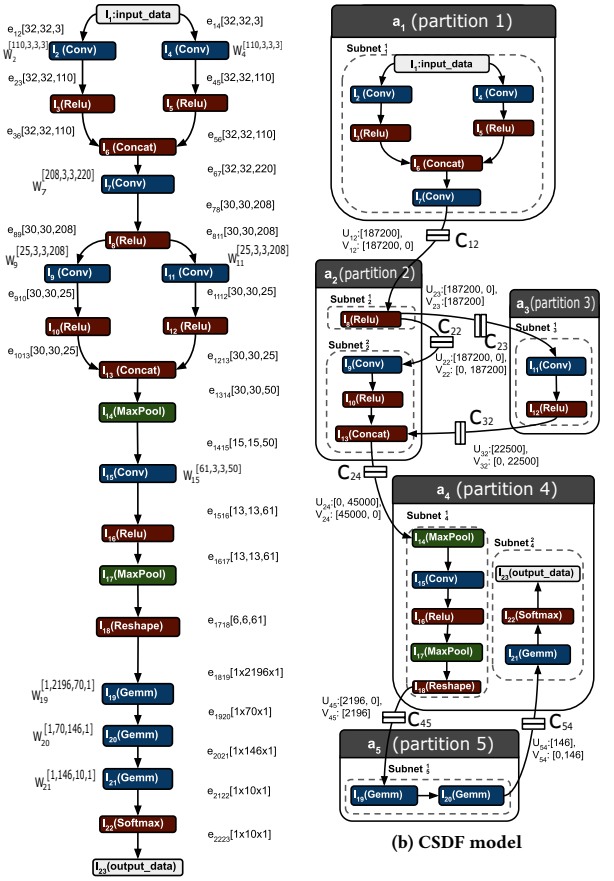


Figure 2: CNN and CSDF computational models

3.1 CNN model

The CNN is a computational model [2], commonly represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers, and a set of edges E. An example of a CNN model with $|L|$=23 layers and $|E|$=24 edges is shown in Figure 2(a). The CNN model specifies the transformations over the CNN input data, e.g., an image, required to obtain the CNN output data, e.g., an image classification result. Every layer $l_i \in L$ specifies a part of these transformations. It has a layer input data X_i, a layer output data Y_i, and an operator op_i, so that $Y_i = op_i(X_i)$. Convolutional layers and fully connected layers also have weights W_i. The operator op_i determines the main difference between the CNN layer types. The most common layer types [3] are:

The CNN layers input data, output data, and weights are stored in multidimensional arrays, called tensors [2]. In this paper, every input data tensor X_i has the format $X_i^{[H_X,W_X,C_X]}$, where H_X is the input data tensor height, W_X is the input data tensor width, and C_X is the number of input data channels. Similarly, every output data tensor Y_i has the format $Y_i^{[H_Y,W_Y,C_Y]}$, where H_Y is the output data tensor height, W_Y is the output data tensor width, and C_Y is the number of output data channels. For convolutional layers and fully connected layers, every weights tensor W_i has the format $W_i^{[C_Y,H_F,W_F,C_X]}$, where C_Y is the number of output data channels, H_F is the filter height, W_F is the filter width, C_X is the number of input data channels. For other types of layers, $W_i^{[C_Y,H_F,W_F,C_X]} = W_i^{[0,0,0,0]}$ because the other types of layers do not need to use/store weights in order to perform a transformation on their input data X_i.

An example of layer l_2 is given in Figure 2(a). Layer l_2 is a convolutional layer. It applies operator *conv* with weights tensor $W_2^{[110,3,3,3]}$ to its input data tensor $X_2^{[32,32,3]}$ and produces output data tensor $Y_2^{[32,32,110]}$. The input data tensor X_i comes to layer l_i from other CNN layers, as specified by the CNN edges $e_{ji} \in E$. Each CNN edge $e_{ji} \in E$, represents a data dependency between layers l_j and l_i, such that $Y_j \subseteq X_i$. An example of a CNN edge is edge e_{12}, shown in Figure 2(a). Edge e_{12} specifies, that output data of layer l_1 is the input data of layer l_2, i.e., $Y_1^{[32,32,3]} = X_2^{[32,32,3]}$.

A CNN can be partitioned and executed in many different ways on a single MPSoC. However, the partitions are not explicitly specified in the CNN computational model. Therefore, the partitioning to perform the CNN model functionality, and the exact communication and synchronization mechanisms between these different partitions are internally determined by the Deep Learning (DL) framework which is utilized, and can vary for different frameworks. For example, the well-known DL frameworks [1, 9] represent the functionality of every CNN layer l_i as multiple partitions, where the total number of partitions depends on the layer mapping. The framework [14] represents the functionality of the same layer l_i as one partition or part of a partition. Therefore, the partitioning strategy is not explicitly specified in the CNN model.

3.2 CSDF model

The CSDF model [5] is a well-known dataflow model of computation, widely used in the embedded systems community for efficient mapping of applications on embedded devices [7], including distributed embedded system. An application, modeled as a CSDF, is a cyclical graph $G(A, C)$, which consists of a set of nodes A, also called actors, communicating through a set of FIFO channels C. The actors A have cyclically changing firing rules, and channels C have cyclically changing production and consumption rates. An example of a CSDF model with $|A|$=5 actors and $|C|$=7 FIFO channels is shown in Figure 2(b). Every actor $a_i \in A$ in the CSDF model performs an execution sequence F_i of length P_i, where $p \in [1, P_i]$ is called a phase of actor a_i. At every phase p, actor a_i executes function $f_i(((p-1) mod P_i)+1)$. An example of CSDF actor a_2 is given in Figure 2(b). Actor a_2 performs execution sequence $F_2 = \{Subnet_2^1, Subnet_2^2\}$, where $Subnet_2^1$ and $Subnbet_2^2$ are functions. Actor a_2 has $P_2 = 2$ phases. At phase $p = 1$ actor a_2 performs

function $f(1) = Subnet_2^1$, and at phase $p = 2$ actor a_2 performs function $f(2) = Subnet_2^2$. Every FIFO channel $c_{ij} \in C$ in the CSDF model represents data dependency and transfers data in tokens between actors a_i and a_j. c_{ij} has a data production sequence U_{ij} and a data consumption sequence V_{ij}. U_{ij} specifies the production of data tokens into channel c_{ij} by actor a_i. V_{ij} specifies the consumption of data tokens from channel c_{ij} by actor a_j. An example of a CSDF communication channel c_{12} is given in Figure 2(b). Channel c_{12} transfers data between actors a_1 and a_2. It has production sequence U_{12}=[187200], specifying, that actor a_1 produces 187200 tokens into channel c_{12} at its single phase $p = 1$, and consumption sequence V_{12}=[187200, 0], specifying, that actor a_2 consumes 187200 tokens from channel c_{12} at phase $p = 1$ and 0 tokens at phase $p = 2$.

When a CSDF modeled application is executed, each actor in the CSDF can be regarded as one partition, as shown in Figure 2(b). Therefore, the CSDF explicitly specifies the partitions as well as the communication (via the FIFO channels) and synchronization (via the changing firing rules) between different partitions and between different phases of the same partition.

4 OUR METHODOLOGY

In this section, we present our novel partitioning strategy, called Vertical Partitioning Strategy, together with our novel methodology needed to utilize our partitioning strategy efficiently, in order to reduce the memory requirement per edge device and increase the overall system performance when CNN inference is performed on fully distributed system at the edge. In Section 4.1, our novel vertical partitioning strategy is introduced and compared with the three main partitioning strategies introduced in Section 1. In Section 4.2 and Section 4.3 our novel methodology, shown in Figure 3, is explained. It consists of two main steps. In Step 1 (Section 4.2), we utilize a Genetic Algorithm (GA) to find an efficient partitioning of a CNN model, based on our vertical partitioning strategy. In Step 2 (Section 4.3), we use the partitioning, obtained in Step 1, to convert the CNN model into a CSDF model, representing the final executable CNN inference application for a fully distributed system at the edge.

4.1 Vertical Partitioning Strategy

In this subsection, we present in detail our novel partitioning strategy, called Vertical Partitioning Strategy, to deploy a CNN model and perform inference on a distributed system at the edge. The main features of our vertical partitioning strategy are shown in Row 6 of Table 1. Our strategy may partition the layers of a CNN model such that each partition (P_i) includes non-consecutive CNN layers. The input data given to a CNN layer (L_i) and its weights are not partitioned. For example, Figure 1(d) shows a CNN model with ten layers ($L_1 - L_{10}$) that is partitioned into four partitions ($P_1 - P_4$). The memory, required to deploy each partition, can be significantly reduced because both the memory needed for storing the data that is exchanged between the layers of a partition and the memory needed for storing the weights of the layers of a partition can be significantly reduced when the number of partitions is large. Moreover, our strategy supports the execution of partitions in a pipeline fashion, which may help to exploit both data-level and task-level parallelism and to increase the system performance.

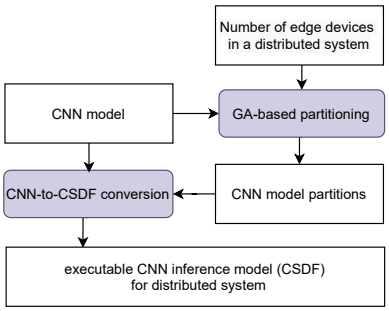


Figure 3: Our methodology

The advantage of our strategy over the Data Partitioning Strategy (Figure 1(a)) is that our strategy gives a better chance to fit a CNN model partition in an edge device, when the CNN model has many layers and/or some layers require huge number of weights to be stored because our strategy reduces both the memory needed to store weights and the memory needed to exchange data between layers of a partition. The advantage of our strategy over the Sequential Partitioning Strategy (Figure 1(b)) is that, our strategy allows to evenly distribute the required memory and computation workload of a CNN model among different partitions because each partition may include non-consecutive CNN layers and has more chances to include diverse layers. As a consequence, the evenly distributed memory per partition may require edge devices with lower memory capacity, and the evenly distributed workload may lead to a higher overall system performance. The advantage of our strategy over the Horizontal Partitioning Strategy (Figure 1(c)) is that, our strategy has more chances to fit a CNN model partition in an edge device, when the CNN model requires huge amount of data to be exchanged between layers because our strategy reduces both the memory needed to store weights and the memory needed to exchange data between layers of a partition. In addition, our strategy helps to increase the overall system performance because it supports the execution of the partitions in a pipeline fashion.

4.2 GA-based Efficient Partitioning

In this subsection, we explain how we obtain an efficient partitioning of a CNN model $CNN(L, E)$ onto a fully distributed system $Devices = \{device_1, device_2, ..., device_n\}$. In our methodology, the CNN model layers L, are distributed onto n partitions, where $n = |Devices|$ is the number of edge devices, available in the distributed system, and each partition is deployed on one edge device. We define a partitioning of CNN model $CNN(L, E)$ onto distributed system $Devices$ as a division of layers set L into n subsets. We denote such partitioning as ${}^nL = \{{}^nL_1, {}^nL_2, ..., {}^nL_n\}$, where each partition ${}^nL_i \in {}^nL$ is a subset of layers, deployed on $device_i$, such that $\cap_{i=1}^{n} {}^nL_i = \emptyset$, and $\cup_{i=1}^{n} {}^nL_i = L$. An example of partitioning ${}^5L = \{{}^5L_1, {}^5L_2, {}^5L_3, {}^5L_4, {}^5L_5\}$ of the CNN model $CNN(L, E)$, shown in Figure 2(a) and explained in Section 3.1, on fully distributed system $Devices = \{device_1, device_2, device_3, device_4, device_5\}$, is given in Table 2. Every Column in Table 2 corresponds to a subset ${}^5L_i, i \in [1, 5]$. For example, Column 1 in Table 2 corresponds to subset ${}^5L_1 = \{l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$. The layers within subset ${}^5L_1$

$device_1$	$device_2$	$device_3$	$device_4$	$device_5$
$l_1, l_2, l_3, l_4, l_5, l_6, l_7$	l_8, l_9, l_{10}, l_{13}	l_{11}, l_{12}	$l_{14}, l_{15}, l_{16}, l_{17}, l_{18}, l_{21}, l_{22}, l_{23}$	l_{19}, l_{20}

Table 2: Partitioning example

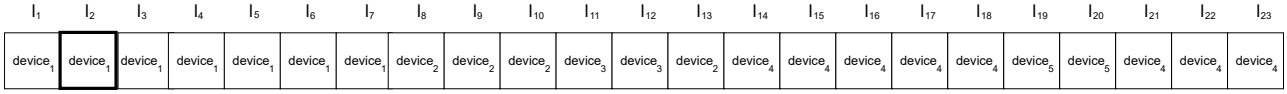


Figure 4: Partitioning chromosome example

are deployed on $device_1$. Column 2 in Table 2 describes subset ${}^5L_2 = \{l_8, l_9, l_{10}, l_{13}\}$. Every layer $l_i \in {}^5L_2$ is deployed on $device_2$.

We consider that a partitioning ${}^nL$ is efficient, if it ensures that the memory required by a CNN model is evenly distributed among all edge devices. We note that obtaining such an efficient partitioning of a CNN model onto a fully distributed system at the edge is a difficult and complex Design Space Exploration (DSE) problem. In our methodology, to solve this problem, we propose to use a Genetic Algorithm (GA) [12]: a well-known heuristic approach, widely used for finding optimal solutions for complex DSE problems. We use a simple GA with standard two-parent crossover, a single-gene mutation, and standard user-defined GA parameters, such as initial offspring size, number of epochs, mutation and crossover probabilities [12]. To utilize such a GA for searching of an efficient partitioning ${}^nL$, we have to specify two problem-specific GA attributes, namely a chromosome and a fitness function [12]. A chromosome is a representation of a GA solution (in our methodology a solution is a partitioning) as a set of parameters (genes), joined into a string [12]. We represent partitioning ${}^nL$, as a string of length $|L|$, where every gene is an edge device $device_i \in Devices$. An example of the chromosome, corresponding to partitioning ${}^5L$, shown in Table 2, is given in Figure 4.

The fitness function is a special function, which evaluates the quality of the solutions and guides the GA-based search. During the search, the fitness function should be minimized or maximized. In our methodology, we search for a partitioning, in which the memory required by a CNN model is evenly distributed among all edge devices, available in $Devices$, i.e., the difference in the memory requirements between every pair of edge devices ($device_i \in Devices, device_j \in Devices, i \neq j$) is minimized. Thus, we define a specific fitness function ϕ to be minimized during the GA-based search as:

$$\phi = \sum_{\forall(device_i, device_j) \in Devices^2} |\beta_{device_i} - \beta_{device_j}| \quad (1)$$

where β_{device_i} and β_{device_j} are the total memory requirement for $device_i$ and $device_j$, respectively. For every $device_i \in Devices$, β_{device_i} is computed as:

$$\beta_{device_i} = \beta^w_{device_i} + \beta^d_{device_i} \quad (2)$$

where $\beta^w_{device_i}$ is the memory needed to store the weights tensors of all layers, deployed on $device_i$; $\beta^d_{device_i}$ is the upper bound of the memory needed to store the data tensors of all layers, deployed on $device_i$. Note that we use the upper bound of the memory to search for the efficient partitioning because the actual memory needed for the data tensors depends on the method of their implementation and the optimizations applied for memory reuse. The memory $\beta^w_{device_i}$ is computed as:

$$\beta^w_{device_i} = \sum_{l_k \in {}^nL_i} (C_{Y_k} \times H_{F_k} \times W_{F_k} \times C_{X_k}) \times \beta_{(token)} \quad (3)$$

where ${}^nL_i$ is the set of all layers, deployed on $device_i$, $C_{Y_k} \times H_{F_k} \times W_{F_k} \times C_{X_k}$ is the size of the weights tensor of layer $l_k \in {}^nL_i$ in data tokens, $\beta_{(token)}$ is the memory needed to store one data token. The memory $\beta^d_{device_i}$ is computed as:

$$\beta^d_{device_i} = \sum_{l_k \in {}^nL_i} (H_{X_k} \times W_{X_k} \times C_{X_k} + H_{Y_k} \times W_{Y_k} \times C_{Y_k}) \times \beta_{(token)} \quad (4)$$

where ${}^nL_i$ is the set of all layers, deployed on $device_i$; $H_{X_k} \times W_{X_k} \times C_{X_k}$ is the size of the input data tensor of layer $l_k \in {}^nL_i$ in data tokens, $H_{Y_k} \times W_{Y_k} \times C_{Y_k}$ is the size of the output data tensor of layer $l_k \in {}^nL_i$ in data tokens, $\beta_{(token)}$ is the memory needed to store one data token.

4.3 CNN to CSDF model conversion

In this subsection, we show how we convert a CNN model, introduced in Section 3.1, into a final executable application, represented as a CSDF model, introduced in Section 3.2, for distributed system at the edge. We utilize the conversion algorithm proposed in [11]. As inputs to this algorithm, we provide a CNN model $CNN(L, E)$ and an efficient partitioning ${}^nL$, obtained by using the GA-based approach in Section 4.2. The algorithm generates a CSDF model $G(A, C)$, which performs the functionality of the CNN model $CNN(L, E)$, efficiently deployed on a distributed system at the edge, as specified by partitioning ${}^nL$. An example of the generated CSDF model $G(A, C)$ from the CNN model $CNN(L, E)$, shown in Figure 2(a) and explained in Section 3.1, with the partitioning ${}^5L$, shown in Table 2 and explained in Section 4.2, is given in Figure 2(b). Every partition ${}^nL_i \in {}^nL$ of the CNN model is represented as an actor $a_i \in A$ in the CSDF model. For example, as shown in Figure 2(b), the five partitions $\{{}^5L_1, {}^5L_2, {}^5L_3, {}^5L_4, {}^5L_5\}$ of the CNN model are represented as the five actors $\{a_1, a_2, a_3, a_4, a_5\}$ of the CSDF model. Every edge $e_{sp} \in E$ between two consecutive layers l_s and l_p of the CNN model that belong to partitions ${}^nL_i$ and ${}^nL_j$ with corresponding

actors a_i and a_j, is represented as a communication FIFO channel $c_{ij} \in C$ in the CSDF model. Every $c_{ij} \in C$ has a data production sequence U_{ij} and a data consumption sequence V_{ij}, where U_{ij} and V_{ij} are determined by the size of the output and input tensors associated with edge e_{sp}. For example, edge e_{78} between layers l_7 and l_8, shown in Figure 2(a), is represented as communication FIFO channel c_{12}, shown in Figure 2(b). c_{12} has a data production sequence $U_{12} = [187200]$ and a data consumption sequence $V_{12} = [187200, 0]$ because the size of tensors ${Y_7}^{[30,30,208]}$ and ${X_8}^{[30,30,208]}$ associated with edge e_{78} is $30 \times 30 \times 208 = 187200$. Every actor $a_i \in A$ represents the functionality of all CNN layers that belong to the CNN partition ${}^nL_i \in {}^nL$ which corresponds to a_i. The execution of an actor a_i can be performed in several phases. At every phase $p \in [1, P_i]$ actor a_i executes function $Subnet_i^p$. Every $Subnet_i^p$ represents the functionality of consecutive layers ${}^nL_i^p \subseteq {}^nL_i$. For example, actor a_4, shown in Figure 2(b), represents the functionality of all CNN layers that belong to partition ${}^5L_4$ which corresponds to a_4. It consists of $Subnets_4 = \{Subnet_4^1, Subnet_4^2\}$. The execution of actor a_4 is performed in two phases because a_4 represents the functionality of layers $\{l_{14}, l_{15}, l_{16}, l_{17}, l_{18}, l_{21}, l_{22}, l_{23}\} \in {}^5L_4$ and these layers cab be divided into two groups of consecutive layers, namely ${}^5L_4^1 = \{l_{14}, l_{15}, l_{16}, l_{17}, l_{18}\}$ and ${}^5L_4^2 = \{l_{21}, l_{22}, l_{23}\}$. Therefore, at phase $p = 1$ actor a_4 executes function $Subnet_4^1$ corresponding to the sequence of layers ${}^5L_4^1$ and at phase $p = 2$ actor a_4 executes function $Subnet_4^2$ corresponding to the sequence of layers ${}^5L_4^2$.

Between some actors, cyclic dependencies occur, that may lead to deadlocks in the CSDF model. To avoid the deadlocks, the conversion algorithm specifies the execution of some actors in more phases, such that at every phase $p \in [1, P_i]$, actor a_i has no cyclic dependencies. For the example, shown in Figure 2(b), a cyclic dependency occurs between actors a_2 and a_3. If actor a_2 would execute layers l_8 and l_{13} in one phase, according to the semantics of the CSDF model [5], it would expect 187200 data tokens to be present in channel c_{12} and 22500 data tokens to be present in channel c_{32}, before it can fire. However, data in channel c_{32}, should be produced by actor a_3, which, before it can fire, expects actor a_2 to produce 187200 data tokens in channel c_{23}. Thus, such execution would lead to a deadlock in the CNN inference. To avoid the deadlock, the conversion algorithm specifies the execution of actor a_2 in 2 phases which are connected by communication FIFO channel c_{22}. At phase $p = 1$, actor a_2 executes only layer l_8. It consumes data only from channel c_{12}, and produces data to channel c_{23}, such that actor a_3 can fire. At phase $p = 2$, actor a_2 consumes data only from channel c_{32}, and executes layers l_9, l_{10} and l_{13}. Thus, at every phase $p = [1, 2]$, actor a_2 has no cyclic dependencies, and no deadlock occurs in the CSDF model execution.

Each actor $a_i \in A$ in the CSDF model is mapped and executed on one edge device $device_i$, and the function $Subnet_i^p$, executed at each phase $p \in [1, P_i]$ of actor a_i, can be specified as a small CNN and implemented by using an existing DL framework (e.g., TensorRT). Such implementation benefits from the optimization methods for CNN inference provided by the existing DL framework and exploits data-level parallelism available within the layers as well. Each communication FIFO channel $c_{ij} \in C$ in the CSDF model is implemented on the two edge devices $device_i$ and $device_j$ that are connected via c_{ij}, in order to evenly distribute the overall memory needed for c_{ij} between edge devices $device_i$ and $device_j$. The communication FIFO channels explicitly specify and implement the communication and synchronization between actors of the CSDF model. Every FIFO channel c_{ij} has at least space for $U_{ij} + V_{ij}$ data tokens in order to ensure that the actors of the CSDF model can be launched in a pipeline fashion and task-level parallelism is exploited. The memory needed to store one data token in the FIFO is determined by the size (in Bytes) of one data element stored in the tensors corresponding to c_{ij}.

5 EXPERIMENTAL RESULTS

In this section, we present experimental results, obtained by using our novel partitioning strategy and methodology, on real-world CNNs from the ONNX models zoo library, implemented onto fully distributed systems that consist of 2 to 10 Nvidia Jetson-TX2 MPSoCs embedded devices. The goal of the experiments is to demonstrate that our novel proposed vertical partitioning strategy and novel methodology are able to deliver CNN inference on a fully distributed system at the edge with a lower memory requirement per edge device and/or higher system performance, compared to the existing methodologies that utilize the other three existing partitioning strategies, introduced in Section 1 and Section 2.

We use three real-world CNNs, namely YOLOv2, Resnet50, and Vgg16 from the ONNX models zoo library that take images as input for CNN inference. These CNNs are utilized in different applications and have diverse number of layers, diverse memory requirements, and diverse performance. Such diversity leads to a diverse scale of memory requirement per edge device and overall system performance when these CNNs are implemented onto fully distributed systems at the edge. YOLOv2 is used for object detection and image segmentation. It has 26 layers, requires 336.2 MB of memory and can achieve a performance of 2.32 frames per second (fps), if implemented on one Nvidia Jetson-TX2 embedded device. Resnet50 and Vgg16 are used for image classification. Resnet50 has 125 layers, requires 298.1 MB of memory and can achieve a performance of 2.13 fps, if implemented on one Nvidia Jetson-TX2 embedded device. Vgg16 has 21 layers, requires 232.8 MB of memory and can achieve a performance of 0.81 fps, if implemented on one Nvidia Jetson-TX2 embedded device. So, these three CNN models are sufficiently representative and good examples to apply our partitioning strategy and methodology on and to demonstrate its merits. Moreover, we can compare our experimental results on YOLOv2 with results obtained by the methodologies that utilize the other three existing partitioning strategies, introduced in Section 1 and Section 2, because they also report experimental results on YOLOv2.

The three CNN models, mentioned above, are distributed efficiently using our partitioning strategy and methodology. The memory requirement per edge device is computed by using Equation 2, introduced in Section 4.2. The GA of our methodology is executed with initial population size =5000, number of generations = 100, mutation probability = 5%. The overall system performance is directly measured on the distributed system, when the CNN models are converted to CSDF models (introduced in Section 4.3) and the different partitions of the CSDF models are mapped and executed on different Jetson-TX2 devices of the distributed system.

Table 3: Experimental results

number of edge devices	YOLOv2		Resnet50		Vgg16	
	memory per device saving rate (%)	performance speedup (times)	memory per device saving rate (%)	performance speedup (times)	memory per device saving rate (%)	performance speedup (times)
2	48.2	1.9	48.1	1.9	47.7	1.9
3	64.1	2.6	63.8	2.7	63.5	2.4
4	68.6	3.1	70.8	3.4	70.2	3.1
5	71.9	3.9	75.3	3.8	70.2	3.1
6	74.3	4.7	79.0	3.9	70.2	3.1
7	74.3	4.7	80.1	4.9	70.2	3.1
8	74.3	4.7	82.1	6.6	70.2	3.1
9	74.3	4.7	84.3	7.5	70.2	3.1
10	74.3	4.7	84.9	7.9	70.2	3.1

The experimental results are shown in Table 3. Column 1 lists the number of edge devices in a distributed system. Columns 2 shows the saving rate in percentage (%) of the memory required per device, compared with the memory needed if the whole CNN model is mapped and executed on 1 device, for the YOLOv2 implementation on a distributed system. Columns 3 shows the overall system performance speedups, compared with a system containing only 1 device, for the YOLOv2 implementation on a distributed system. Similarly, Columns 4-5 show the experimental results for Resnet50, and Columns 6-7 show the experimental results for Vgg16. From Table 3, we can see that, the saving rate of the memory required per device and the overall system performance speedup do not increase when the number of edge devices is more than 6 for YOLOv2, and the number of edge devices is more than 4 for Vgg16. The reason is that when one CNN partition consists of only one CNN layer with a huge memory requirement, the saving rate of the memory required per device and the overall system performance speedup will be determined by this particular bottleneck CNN layer because our partitioning strategy does not split individual CNN layers. However, when a CNN model has many layers, for example Resnet50 in Table 3, then the saving rate of the memory required per device and the overall system performance speedup have more potential to keep increasing, with the increase of the number of edge devices in a distributed system, because our methodology has more options to split the CNN model into multiple partitions where a single layer is not the bottleneck in the system.

From the experimental results for YOLOv2, we can conclude that our proposed partitioning strategy and methodology are able to deliver CNN inference on a fully distributed system at the edge with lower memory requirement per edge device and/or higher overall system performance, compared to the methodologies that utilize the other three existing partitioning strategies, as explained in detail and shown in Section 2. From the experimental results for Resnet50 and Vgg16, we can conclude that our proposed partitioning strategy and methodology can be utilized efficiently on other representative CNNs for inference on distributed systems delivering CNN inference with very reduced memory requirement per edge device and improved overall system performance at the same time.

6 CONCLUSIONS

In this paper, we discuss the advantages and disadvantages of existing methodologies utilizing different partitioning strategies, namely Data Partitioning Strategy, Sequential Partitioning Strategy, and Horizontal Partitioning Strategy, for CNN model inference on fully distributed system at the edge. In order to avoid the disadvantages of the aforementioned strategies in terms of memory requirement per device and overall system performance, we propose a novel partitioning strategy, called Vertical Partitioning Strategy, which reduces both the memory needed to store weights and the memory needed to exchange data between layers of a partition, and also enables even distribution of the required memory and computation workload of a CNN model among different partitions. At the same time, our strategy helps to increase the overall system performance because it supports the execution of the partitions in a pipeline fashion which exploits both data-level and task-level parallelism. Moreover, we propose a novel methodology needed to utilize our partitioning strategy efficiently. The experimental results for YOLOv2 show that our proposed partitioning strategy and methodology are able to deliver CNN inference on a fully distributed system at the edge with lower memory requirement per edge device and/or higher overall system performance, compared to existing methodologies that utilize the aforementioned existing partitioning strategies. The experimental results for Resnet50 and Vgg16 show that our proposed partitioning strategy and methodology can be utilized efficiently on other representative CNNs for inference on distributed systems delivering CNN inference with very reduced memory requirement per edge device and improved overall system performance at the same time.

REFERENCES

[1] M. Abadi, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. (2016).

[2] M. Abadi, et al. 2017. A computational model for TensorFlow: an introduction. In *1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.

[3] M. Alom, et al. 2018. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164* (2018).

[4] et al. S. Bhattacharya. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*.

[5] G. Bilsen, et al. 1996. Cycle-static dataflow. *IEEE Transactions on signal processing* (1996).

[6] Y. Cheng, et al. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[7] S. Ha, et al. 2017. *Handbook of hardware/software codesign.* Springer,.

[8] et al. S. Han. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. (2015).

[9] Y. Jia, et al. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*.

[10] et al. J. Mao. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE.

[11] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *International Conference on Embedded Computer Systems*. Springer.

[12] K. Sastry, et al. 2005. Genetic algorithms. In *Search methodologies*. Springer.

[13] R. Stahl, et al. 2019. Fully distributed deep learning inference on resource-constrained edge devices. In *International Conference on Embedded Computer Systems*.

[14] S. Wang, et al. 2019. High-throughput CNN inference on embedded ARM Big. LITTLE multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[15] et al. S. Yao. 2017. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*.

[16] Z. Zhao, et al. 2018. DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).