Fully Distributed Deep Learning Inference on Resource-Constrained Edge Devices *

Rafael Stahl[1], Zhuoran Zhao[2], Daniel Mueller-Gritschneder[1], Andreas Gerstlauer[2], and Ulf Schlichtmann[1]

[1] Technical University of Munich, Germany
{r.stahl,daniel.mueller,ulf.schlichtmann}@tum.de
[2] University of Texas at Austin, USA zhuoran@utexas.edu,gerstl@ece.utexas.edu

Abstract. Performing inference tasks of deep learning applications on IoT edge devices ensures privacy of input data and can result in shorter latency when compared to a cloud solution. As most edge devices are memory- and compute-constrained, they cannot store and execute a complete Deep Neural Network (DNN). One possible solution is to distribute the DNN across multiple edge devices. For a complete distribution, both fully-connected and feature- and weight-intensive convolutional layers need to be partitioned to reduce the amount of computation and data on each resource-constrained edge device. At the same time, resulting communication overheads need to be considered. Existing work on distributed DNN execution can not support all types of networks and layers or does not account for layer fusion opportunities to reduce communication. In this paper, we jointly optimize memory, computation and communication demands for distributed execution of complete neural networks covering all layers. This is achieved through techniques that combine both feature and weight partitioning with a communication-aware layer fusion approach to enable holistic optimization across layers. For a given number of edge devices, the schemes are applied jointly such that the amount of data to be exchanged between devices is minimized to optimize run time.

Experimental results for a simulation of six edge devices on 100 Mbit connections running the YOLOv2 DNN model show that the schemes evenly balance the memory footprint between devices. The integration of layer fusion additionally leads to a reduction of communication demands by 14.8%. This results in run time speed-up of the inference task by 1.15x compared to partitioning without fusing.

Keywords: Deep Learning · Distributed Computing · IoT

* This work is partly funded by National Science Foundation (NSF) grant NSF CNS-1421642 in the USA and the German ministry of education and research (BMBF) under grant number 01IS17028F as part of the ITEA3 project COMPACT with reference number 16018.

1 Introduction

In the context of the Internet of Things (IoT), deep learning has emerged as a valuable tool to process complex or noisy sensory input at high quality. A Deep Neural Network (DNN) is trained upfront with high computational effort with a large input data set. After training is completed, the DNN can be given previously unseen input data and will, e.g., return a classification result. The latter task is called inference and is the focus of this paper. Inference is also a resource-intensive operation since a large amount of input or intermediate data and trained weights have to be stored and many computational operations are required. Therefore, it is not possible to perform this task on a single small edge device. While this could be solved by deploying more powerful edge devices, the cost of such a solution is prohibitive. Another possible solution is off-loading of the inference to powerful servers in the cloud. However, this approach introduces privacy issues concerning the input data and requires high bandwidth to the cloud [3]. Other approaches shrink and prune the total size of the DNN, but this reduces the model's accuracy and might make it no longer viable for a given problem [2,7].

An orthogonal solution is the utilization of multiple edge devices to cooperate on executing the inference task. In many IoT applications, a large number of edge devices is available and connected with each other via some local network. When inputs arrive rarely, most devices are idle. A device receiving an input can use the idle time of the other edge devices to solve the inference task offering a low-cost but efficient solution. Existing approaches for distributed execution of DNN inference tasks across clusters of edge devices, however, often only consider a subset of DNN layers and still require a powerful gateway device to execute other parts of a network [11]. Other approaches [6] are able to distribute an entire network, but have limitations in the type and size of layers that can be fitted, and they do not consider opportunities to improve communication demands further.

In this paper, we investigate methods for distributed execution of complete DNNs covering all types of layers while jointly optimizing for computation, memory and communication demands. We build on our prior work [11] that introduced an approach for memory- and communication-aware partitioning and fusing of feature-dominated early convolutional layers. We extend this approach with novel methods to partition convolutional and fully-connected layers whose weight data size dominates their input and output data size. This allows partitions for all layers to be executed in a parallel fashion on several small edge devices to enable a fully distributed inference. The computation and memory footprint of processing and storing feature and weight data is evenly distributed over all devices, such that the DNN inference task can be scaled down for any size of IoT edge device. A major problem with partitioning is that the inputs and outputs of these layers need to be communicated from one edge device to the other. We further present a new scheme to minimize this communication overhead by enabling fusing across all types of layers. In detail, the contributions of this paper are:

1) A memory- and communication-aware partitioning scheme for fully-connected and weight-intensive convolutional layers that, when combined with prior schemes on feature-intensive convolutional layer partitioning, enables complete distribution of arbitrary state-of-the-art DNNs.

2) A new method that can fuse fully-connected or convolutional layers such that each fused layer partition can be processed on a single device without the need to communicate intermediate data between layers with other edge devices.

3) An integer linear programming (ILP) formulation to identify the best partitioning and fusing scheme for a given DNN. The scheme minimizes the communication demand of exchanging input and output data between devices. This leads to a significant reduction in inference run time, which strongly depends on communication time.

Experimental results show that the memory footprint is scaled down evenly with the number of available edge devices. Optimizing the partitioning schemes with layer fusion leads to 14.8% reduced communication demand, while executing the inference task for the YOLOv2 DNN on six edge devices with 100 Mbit connections. This results in a run time speed-up by 1.15x compared to only partitioning the outputs.

2 Related Work

Fully distributed inference is tackled by MoDNN [6]. MoDNN distributes a DNN across multiple mobile phones connected via a wireless network. The approach also targets the distribution of both the layer input and output data as well as the weight data across devices. While they are able to partition weights, they use an approach that focuses heavily on sparse fully-connected layers (i.e., fully-connected structures, where some weights are zero). It is not optimized for communication in dense layers, and weight-intensive convolutional layers are not addressed. Furthermore, they process networks in a layer-by-layer fashion, which requires all devices to synchronize and exchange data after each layer. Their method could be combined with the layer fusion proposed in this work to minimze communication overhead after adding additional constraints.

Other works approach the execution of an inference task on edge devices from a different angle by pruning and quantizing the model [2,7]. Yet another line of work has focused on distributing different parts of a model to different tiers of processing power with the possibility of early exits that would make them edge-only [10]. Both of these are orthogonal methods to the one proposed here.

The distribution of the model was also tackled in a different context for hardware acceleration of DNNs in [1]. The central idea of that work is to fuse the first few layers of the network in order to reduce total data transfer to and from the chip. In that work, the memory-constrained device is the accelerator instead of an IoT edge device. Fusing optimizations for the accelerator are only investigated for the first layers. By contrast, the fusing approach presented in this paper targets the later network layers, for which the weight data dominates. Both techniques can possibly be combined.

The work in this paper builds on our prior work on adaptive distributed deep learning inference in clusters with dynamic availability of edge nodes [11]. This prior work presented a fusing approach for multiple layers with focus on data partitioning in the first layers. However, it does not consider weight partitioning. At some depth in the DNN, it is no longer possible for that approach to store the large weight data on a single device. This requires later layers to be evaluated on a central powerful gateway edge device that has sufficient memory. These constraints are removed by the approach presented in this work, enabling a fully distributed inference on a set of memory-constrained edge devices.

3 Background on DNNs

The basic DNN inference structure for image recognition can be described as follows: The input of the DNN is an image with each pixel of each color channel being represented by a neuron. It is processed by a number of convolutional layers that apply multiple learned filter functions to their layer input to produce a set of feature maps with each feature map corresponding to a learned filter. These feature maps should capture different characteristics of the image. With a high number of filters in each layer, the number of feature maps typically grows with each convolutional layer. Therefore, aside from convolutional layers there are also pooling layers, which shrink the width and height of feature maps in order to allow a higher number of features to be extracted. The number of intermediate feature maps increases while they loose resolution. As a result, the input and output data dominate the memory usage for the first layers of a DNN, and in the later layers the weights dominate. These later layers are in the focus of this work. Distribution strategies for early layers are described in our prior work [11].

If the DNN has just the task of classifying what it sees, the last few layers typically consist of fully-connected layers as seen in AlexNet [4] and VGGNet [9]. Another possible task is object localization in the image along with classification of multiple objects. In that case the last few layers will also be convolutional and the network is called a Fully Convolutional Network (FCN) [5]. YOLOv2 [8] is used to evaluate our proposed layer fusion method. YOLOv2 is a FCN that follows the typical data size distribution described above.

3.1 Fully-Connected Layers

A fully-connected layer has the distinctive property that all input neurons are connected to all output neurons. Its operation can be expressed as:

$$b_{k,l} = f\Big(\sum_{m=1}^{M_l} a_{m,l} \cdot w_{m,k,l}\Big), \quad k \in \{1, \ldots, K_l\} \tag{1}$$

where for the l-th layer, $a_{m,l}$ is the m-th element of the vector of input neurons $\mathbf{a}_l \in \mathbb{R}^{M_l}$, $b_{k,l}$ is the k-th element of the vector of output neurons $\mathbf{b}_l \in \mathbb{R}^{K_l}$,

$w_{m,k,l}$ is the m,k-th element of the weight matrix $\mathbf{W}_l \in \mathbb{R}^{M_l \times K_l}$ and f is the nonlinear activation function.

For fully-connected layers, the number of weights Q_l that have to be stored determines mainly the memory demand of the inference task. Biases can also be considered weights, but since they are comparably negligible in size, they are not considered for distribution in this work. The computation time is dominated by the number of multiplication operations R_l. For a fully-connected layer, we obtain:

$$Q_l = R_l = K_l \cdot M_l \tag{2}$$

3.2 Convolutional Layers

The input to a convolutional layer are multiple two-dimensional feature maps. For example, an input image is represented as three feature maps, one map for each color channel. Multiple image filters are applied that take all input feature maps into consideration by applying classical image filtering to each of them with a two-dimensional kernel. The resulting images of a single filter are all added up to a single output feature map. Since there are usually multiple filters, the output of convolutional layers are multiple feature maps. The layer's operation can be expressed as:

$$\mathbf{B}_{o,l} = f\Big(\sum_{c=1}^{C_l} \operatorname{corr}(\mathbf{A}_{c,l}, \mathbf{W}_{c,o,l})\Big), \quad o \in \{1, \ldots, O_l\} \tag{3}$$

where the matrix $\mathbf{A}_{c,l}$ is the c-th input feature map of the input tensor $\mathbf{A}_l \in \mathbb{R}^{X_l \times Y_l \times C_l}$, the matrix $\mathbf{B}_{o,l}$ is the o-th input feature map of the output tensor $\mathbf{B}_l \in \mathbb{R}^{X_l \times Y_l \times O_l}$ and the matrix $\mathbf{W}_{c,o,l}$ is the kernel connecting the c-th input feature map with the o-th output feature map. The kernels are elements of the four-dimensional weight tensor $\mathbf{W}_l \in \mathbb{R}^{U_l \times V_l \times C_l \times O_l}$. The activation function is again f. The function $\operatorname{corr}(\mathbf{A}, \mathbf{W})$ computes the two-dimensional cross-correlation, for which the x,y-th element is computed with:

$$\operatorname{corr}(\mathbf{A}, \mathbf{W})_{x,y} = \sum_{(u=-\lfloor \frac{U}{2} \rfloor)}^{\lfloor \frac{U}{2} \rfloor} \sum_{(v=-\lfloor \frac{V}{2} \rfloor)}^{\lfloor \frac{V}{2} \rfloor} a_{x+u,y+v} w_{u,v} \tag{4}$$

The size of the input tensor $\mathbf{A}_l$ is $M_l = X_l \cdot Y_l \cdot C_l$, the size of the output tensor is $K_l = X_l \cdot Y_l \cdot O_l$. The number of weights Q_l to be stored and the number of multiplication operations R_l for a convolutional layer, can be given as:

$$Q_l = U_l \cdot V_l \cdot C_l \cdot O_l, \quad R_l = X_l \cdot Y_l \cdot U_l \cdot V_l \cdot C_l \cdot O_l \tag{5}$$

Convolutional layers of DNNs are very heavy on computational and memory resources, requiring to distribute these layers, if only constrained edge devices are available.

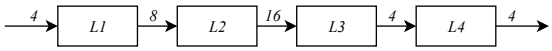

Fig. 1. 4-Layer example on a single device

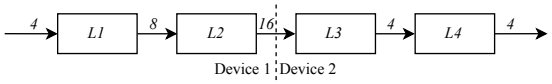

Fig. 2. 4-Layer example with Sequential Layer Mapping

3.3 Distributed Inference

As was already pointed out, for fully-connected layers and later convolutional layers, the number of weights stored in the weight matrix and weight tensor dominate the memory demand. Hence, we approximate the memory footprint F_n with the number of weights stored on the device n. The computational effort is dominated by the multiplications with the weights. Computation on the devices can be parallelized for distributed inference, where T denotes the number of sequential multiplications on the longest path when executing the layers in parallel. As the devices are bandwidth-constrained, the overall run time of the inference task also depends strongly on the communication load C to exchange input or output data between the devices. Overall, larger number of devices leads to lower memory footprint F_n per device, more communication demand C and more parallelism (lower T), where changes in C and T impact run time in opposing ways. Considering that we map a network with L layers to a single device we obtain:

$$F_n^{(N)} = \sum_{l=1}^{L} Q_l, \quad T^{(N)} = \sum_{l=1}^{L} R_l, \quad C^{(N)} = 0 \tag{6}$$

This is illustrated for a fully-connected 4-layer example in Fig. 1, where the edges are annotated with the input/output sizes of the layer. For this example, we obtain: $F_1^{(N)} = 4 \cdot 8 + 8 \cdot 16 + 16 \cdot 4 + 4 \cdot 4 = 240$, $T^{(N)} = 240$ and $C^{(N)} = 0$.

A straight-forward solution for reducing F_n is to distribute different layers to different devices also known as layer pipelining. Using again the above example for two available devices, we can map layers 1 and 2 to device 1 as well as layers 3 and 4 to device 2 as shown in Fig. 2. Here we obtain intuitively: $F_1^{(DL)} = 4 \cdot 8 + 8 \cdot 16 = 160$, $F_2^{(DL)} = 16 \cdot 4 + 4 \cdot 4 = 80$, $T^{(DL)} = 240$ and $C^{(DL)} = 16$. The memory footprint is determined by the larger set of layers as $F_1^{(DL)}$, because the weights are not distributed evenly across the two devices. Moreover, this method does not utilize any parallelism for a single input, leading to the same high run time as for running on a single device. This mapping can be improved by using layer partitioning as proposed in this paper in the following.

4 Layer Partitioning Methods

The central idea of our approach is to apply layer partitioning on the DNN with the objective to evenly distribute the weight data and computational load across

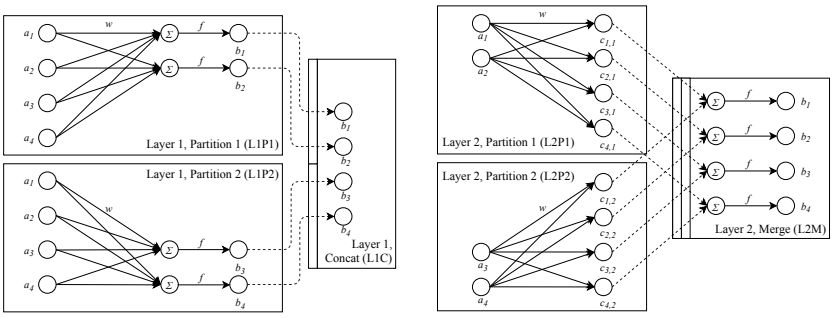

Fig. 3. LOP (left) and LIP (right)

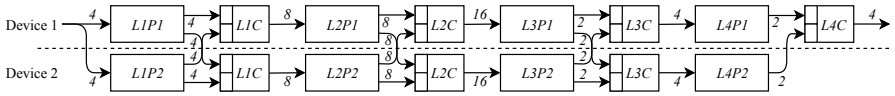

Fig. 4. 4-Layer Example with Layer Output Partitioning

all available devices while minimizing the inference latency. A partitioning can be achieved by splitting either the inputs or the outputs of the layers and mapping one partition of each layer to each device. Input and output partitioning is simple and very effective to distribute the weight data while minimizing communication. In the following, we first discuss these partitioning schemes for the case of fully-connected layers, for which, to the best of our knowledge, such communication-aware solutions were not described yet. As new contributions, we show how partitioning schemes for fully-connected layers can be further improved in terms of communication demand by applying a newly proposed layer fusing scheme. Additionally, we extend all partitioning and fusing schemes to weight-intensive convolutional layers, which were not addressed by existing work before.

4.1 Layer Output and Input Partitioning (LOP and LIP)

In Layer Output Partitioning (LOP) as illustrated in Fig. 3 on the left, we use all of the input $\mathbf{a}_l$ and a part of $\mathbf{W}_l$ to calculate only a subset of output neurons $\mathbf{b}_l$. These outputs can then be fully finalized by applying the activation function f. The final output values then only have to be concatenated ($Concat$) to obtain the full output vector $\mathbf{b}_l$.

Assuming that we use LOP on all L layers for N devices, we obtain the memory footprint $F_n^{(LOP)}$, the execution time $T^{(LOP)}$ and the communication demand $C^{(LOP)}$ as follows:

$$F_n^{(LOP)} = \sum_{l=1}^{L} Q_l \cdot \frac{1}{N}, \quad T^{(LOP)} = \sum_{l=1}^{L} R_l \cdot \frac{1}{N}, \tag{7}$$

$$C^{(LOP)} = \sum_{l=1}^{L} C_l^{(LOP)} = \sum_{l=1}^{L} \left((M_l - o_{l-1}\frac{M_l}{N}) \cdot (N-1) + K_l \cdot \frac{N-1}{N} \right)$$

The inputs have to be fully distributed to the other $(N-1)$ devices, while the outputs only have to be partly communicated. The boolean variable o_l is 1 if the layer l is using LOP. Here it is representing reuse of previous data.

Applying LOP to all four layers of our example as illustrated in Fig. 4, we obtain: $F_1^{(LOP)} = F_2^{(LOP)} = T^{(LOP)} = \frac{240}{2} = 120$ and $C^{(LOP)} = 4+4+4+8+8+2+2+2 = 34$ (arrows crossing between devices). It also shows how previous data can be reused by Device 2. The memory footprint is evenly balanced and we can make full use of parallelism, while requiring some communication.

The second method shown in Fig. 3 on the right is Layer Input Partitioning (LIP). Here, only a part of $\mathbf{a}_l$ is used to calculate incomplete output values for $\mathbf{b}_l$. The number of weights required per device remains unchanged compared to LOP. Because these output values are incomplete, they have to be summed up, before being passed on to the activation function f. A merge operation does the summation and activation.

Assuming that we use LIP on all l layers for N devices, we obtain the same memory footprint $F_n^{(LIP)} = F_n^{(LOP)}$, and the same execution time $T^{(LIP)} = T^{(LOP)}$ as for LOP. The communication demand is:

$$C^{(LIP)} = \sum_{l=1}^{L} C_l^{(LIP)} = \sum_{l=1}^{L} \left(M_l \cdot \frac{N-1}{N} + K_l \cdot (N-1) \right) \tag{8}$$

Here, the outputs have to be fully distributed to the device performing the merging, while the inputs only have to be partly communicated and there is no opportunity for data reuse. This results in large communication overhead when LIP is used by itself. Applying LIP to the example would result in $C^{(LIP)} = 48$.

It is possible to partition the weights in other ways, but this will necessarily increase the required input or output size compared to strict LOP or LIP. For sparse weight data, only the non-zero weight data has to be distributed evenly.

4.2 Fused Layer Partitioning (FUSE)

Every *Concat* or *Merge* operation uses data from all partitions and is therefore a synchronization point preventing parallelism. When combining both LOP and LIP, we achieve layer fusion that eliminates one such synchronization point. The combined operation is shown in Fig. 5. The operation uses LOP on the first layer, but instead of applying the concatenation operation on the partial outputs $\mathbf{b}_l$, LIP is now performed on its following layer. Since the intermediate values c are partial outputs and cannot be passed to the activation function before being summed up, there can be only exactly two consecutive layers that are fused by this method.

Not all layers of the network need to be fused and for an uneven number of consecutive layers this is not even possible. Layers that cannot be fused, can still fall back to LOP or LIP to fulfill memory constraints. Considering this, we obtain the same memory footprint $F_n^{(FUSE)} = F_n^{(LOP)}$, and the same execution time $T^{(FUSE)} = T^{(LOP)}$. Given a certain network of L layers, we define $o_l = 1$

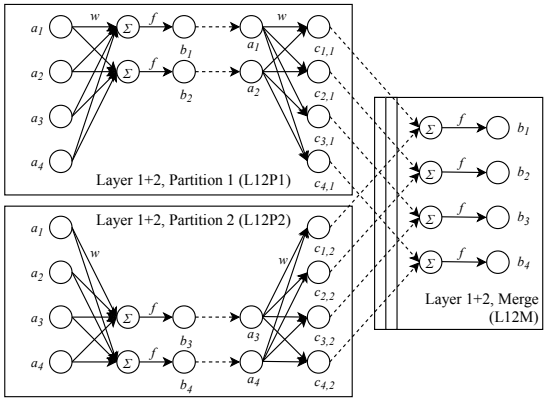

Fig. 5. Fused Layer Partitioning

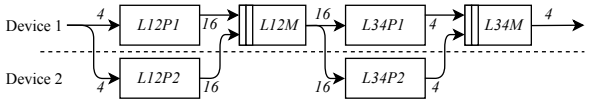

Fig. 6. 4-Layer Example with Fused Layer Partitioning

if layer l is using LOP, $i_l = 1$ if it is using LIP, $f_l = 1$ if it is the first layer of a fused layer and $s_l = 1$ if it is the second layer of a fused layer, otherwise all variables are zero. With this we can formulate the communication demand C as:

$$C_l^{(FUSE1)} = M_l \cdot (N-1), \quad C_l^{(FUSE2)} = K_l \cdot (N-1) \tag{9}$$

$$C^{(FUSE)} = \sum_{l=1}^{L} \left(o_l C_l^{(LOP)} + i_l C_l^{(LIP)} + f_l C_l^{(FUSE1)} + s_l C_l^{(FUSE2)} \right) \tag{10}$$

Here, the outputs of the first layer and the inputs of the second layer do not need to be communicated.

When applying Fused Layer Partitioning with two fusions to our example as shown in Fig 6, we get: $C^{(FUSE)} = 4 + 16 + 16 + 4 = 40$ while F and T remain unchanged compared to LOP. This is a worse result than LOP, but this is because two fusions are not a good solution for the given example.

4.3 ILP for Partitioning and Fusing Decision

Fusing at every possible opportunity as done in Fig. 6 can possibly be an inferior solution to carefully selecting which layers to fuse. The saved communication is dependent on the output and input layer sizes, which can vary greatly between layers. Furthermore, layers can only be fused pairwise. If a layer's output or input is fused, its input or output can in turn no longer be fused and has to be communicated. Additionally, a non-fused layer may either favor LOP or LIP partitioning schemes. We propose to optimize the partitioning and fusing decision

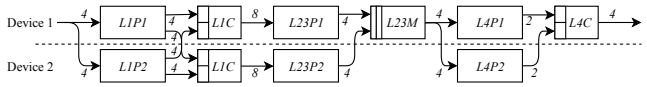

Fig. 7. 4-Layer Example with Optimized Fusion Decision

by minimizing the communication demand C as follows:

$$
\begin{aligned}
min_{\mathbf{o,i,f,s}} \;& C^{(FUSE)} \\
s.t. \;& \forall_{l=1\ldots L} \quad o_l + i_l + f_l + s_l = 1 \quad o_l, i_l, f_l, s_l \in \{0,1\} && (11)\\
& \forall_{l=1\ldots L-1} \quad s_{l+1} = f_l && (12)\\
& f_L = 0 && (13)
\end{aligned}
$$

for which (11) assures that only one partitioning scheme is chosen per layer, (12) assures that the second fused layer is right after the first fused layer and (13) prohibits that the last layer is the first layer of a fusion.

Optimizing the partitioning and fusing decision for our running example leads to the solution shown in Fig. 7. LOP is used for layer 1 and 4 and layer 2 and 3 are fused. A significant reduction is achieved with $C^{(FUSE)} = 22$. This is because the largest communication demand is between layers 2 and 3, which is removed by the fusing decision. Hence, the ILP considers the input and output sizes of all layers to decide on the best partitioning scheme of each layer. The value of C of course is only a rough indicator for the communication impact on a real DNN inference implementation, but our experimental results show that C correlates very well with the run time of the inference task.

4.4 Partitioning of Convolutional Layers

The proposed layer partitioning schemes can also be applied to two consecutive convolutional layers as illustrated in Fig. 8. The figure assumes that three of the feature maps are inside one partition and the data required for one single partition is highlighted. Here, the output feature maps of the first layer are partitioned instead of the output neurons. The first layer only applies a subset of the filters and produces, hence, only parts of the intermediate feature maps. Mathematically it is equivalent to implementing Eq. 3 only for a subset of the o indices assigned to this device. The device only needs to store its own filter weights.

In the second layer, all of the filters are applied to each of the input feature maps. However, only the filter channels corresponding to those partial inputs are required. The partially known input feature maps are correlated and summed up, resulting in partial outputs for all output feature maps. This implements Eq. 3, but now only for a subset of the c indices.

The nonlinear activation function cannot be applied on these partial outputs, because the total sum over c is incomplete. Hence again a merge step is required to sum up the partial results and evaluate the activation function. The equations derived for F_n, T and C for the fully-connected layers are also valid for the

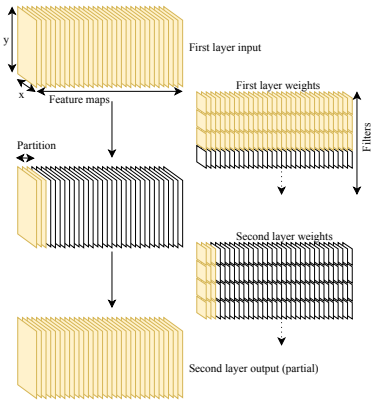

Fig. 8. Convolution weight partitioning

convolutional layers as we used the same symbols M_l, K_l, Q_l and R_l to denote the important parameters for both layers. As for fully-connected layers, the number of weights per device F_n and the number of multiplications T can be divided evenly by the number of devices. This is especially useful because the memory footprint to store the weight tensors is often very high for later convolutional layers of the DNN.

5 Experimental Results

We implemented a fully distributed inference approach using the layer partitioning methods on a set of edge devices on top of the framework from [11]. Edge devices fulfill two different roles during the inference process. Either they provide the input data as the source device, or they assist with the inference as worker devices. The work in [11] delegates all processing of the weight-dominated later layers of the DNN to a single powerful central gateway device. We adapt the framework such that the source device collects the initial results from the early DNN layers for further distribution. The prior existing communication pattern did not require long-running connections between devices. However, with layer fusion, all output data has to be synchronized at least every two layers between worker devices. This causes a large number of messages to be exchanged between source and worker devices. For the overall run time, it is essential that the connections between the source device and all worker devices remain open, which is enabled in our extended framework. The two partial convolution operations that implement LOP and LIP were implemented with the same linear algebra functions as the baseline convolution. To implement layer fusion, these operations were combined into a single operation, for which all communication and memory copies were stripped. These steps extended the existing framework to enable a fully distributed deep learning inference with support for layer fusion.

For evaluation, an emulation setup is chosen over physical devices in order to have accurate control over the edge device and network properties. The emula-

tion host is a Linux desktop computer with a quad-core processor and 16 GB of RAM. For each emulated edge device a new containerized Docker environment is created and all emulated edge devices are connected via a bridged virtual network. This setup allows for each container to be individually configured in terms of processor, memory and network capabilities. Incoming and outgoing bandwidth was limited with the Linux tool `tc`.

YOLOv2 [8] is used to evaluate our proposed layer fusion method. For each measurement result, three simulations were conducted and the results were averaged to take run time variability into account.

5.1 Evaluation of LOP vs. Layer Fusion

Bandwidth			Number of Devices					Run time [s]			
10 Mbit	100 Mbit	1 Gbit	1	3	5	6	7	LOP	FUSE	Speed-up of FUSE vs. LOP	Mem. F_n [MB]
✓						✓		74.2	64.2	1.16x	34
	✓					✓		11.6	10.1	1.15x	34
		✓				✓		5.92	5.59	1.06x	34
	✓		✓					13.1	13.3	0.99x	204
	✓			✓				8.64	8.26	1.05x	68
	✓				✓			10.5	9.35	1.12x	41
	✓						✓	12.1	11.0	1.10x	29

Table 1. Run time and memory footprint F_n comparison for YOLOv2

Table 1 shows the total inference time and memory footprint for different edge device and network parameters for LOP alone vs. Layer Fusion (FUSE). As expected, when increasing the number of devices we observe a proportional decrease of F_n, which allows to scale the inference task for the available memory resources of the edge devices equally for LOP and FUSE.

Existing work is not able to partition weight-dominated convolutional layers. Hence, we can only compare to a baseline that executes on a single node. Here, some reduction in run time can be seen when using other idle nodes for the inference task. There are several effects that influence the final run time of the inference task compared to the theoretical values of T and C when using different numbers of edge devices. First, there is some latency that has to be added for every message between devices. Also, the early layers of the YOLOv2 network are included to see the full run time. Hence, the run time varies when increasing the number of devices. Communication and parallelism have opposing impacts on run time, so that no clear trend is observed when going from three to seven devices.

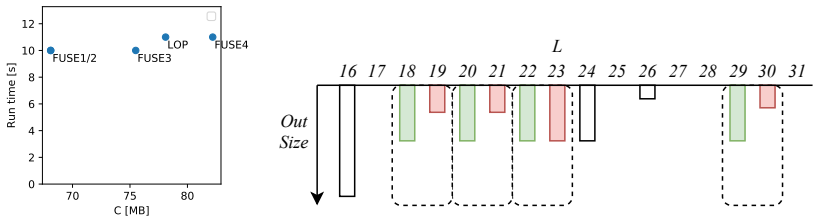

Fig. 9. Fusion decisions for YOLOv2 (left) and decision FUSE1 (right)

When comparing layer fusion to applying LOP only, a clear benefit can be seen in terms of run time. The benefit is lower, when the available bandwidth is very high (1 Gbit) as the Fusing only improves communication demand. The speed-up of FUSE is generally higher, the more number of devices are involved, because there is more communication happening which can be positively affected by layer fusion. Yet, also here some variance is observed. But clearly, it can be seen that layer fusion achieves a significant speed-up without imposing additional costs.

MoDNN [6] does not strictly adhere to the LOP or LIP schemes, so their layers cannot be directly fused. However, fusing can be supported by adjusting the first layer to follow LOP and the second layer to follow LIP. This will not be a restrictive constraint on their method, because they focus on sparse networks. So to cluster complete LOP or LIP nodes, only few connections need to be moved.

5.2 Optimization of Partitioning and Fusing Decisions

To evaluate the ILP optimization for finding an optimal partitioning and fusing decisions, several alternatives in the YOLOv2 model were simulated for 100 Mbit bandwidth and six devices. Table 2 shows which layers were fused for all possible variants. Solving the ILP results in the optimized solutions FUSE1 and FUSE2. Fig. 9 on the left shows the communication size C and the measured run time. As can be seen, there is a good correlation between C and the run time, indicating that the solution of the ILP can give a good fusing decision for DNNs. Yet, the ILP needs to be evaluated for more networks to strengthen these results.

Variant	Fused Layers	C [MB]	C Saving [%]
LOP	none	78.1	0.00
FUSE4	19+20, 21+22, 23+24, 29+30	82.2	-5.00
FUSE3	18+19, 21+22, 23+24, 29+30	75.5	3.60
FUSE2	18+19, 20+21, 23+24, 29+30	68.1	14.8
FUSE1	18+19, 20+21, 22+23, 29+30	68.1	14.8

Table 2. Communication demand for fusion decisions for YOLOv2

Table 2 also shows the communication demand and the savings in communication compared to LOP. For the optimized fusing alternatives, there is a communication demand reduction of 14.8%.

An optimized fusing decision FUSE1 is illustrated in Fig. 9 on the right with the considered layers of the YOLOv2 network and their respective output sizes. The green bars are the output sizes of the first fused layers, which do not need to be communicated due to fusion, while the red ones are the output sizes of the second layers, for which all partial outputs need to be communicated, hence increasing communication size compared to LOP. As can be seen layers with large output size are selected as first layers.

6 Summary and Conclusions

In this paper, an approach for partitioning and fusing consecutive weight-intensive fully-connected or convolutional DNN layers targeting memory-constrained edge devices was presented. This method allows a complete DNN application to be executed in a fully distributed manner on resource-constrained edge clusters. In the process, it reduces communication overhead and therefore improves inference run time. Additionally, an ILP optimization is given, so that optimal fusing decisions can be made.

References

1. Alwani, M., Chen, H., Ferdman, M., Milder, P.: Fused-layer CNN accelerators. In: MICRO (2016)
2. Bhattacharya, S., Lane, N.D.: Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: SenSys (2016)
3. Kang, Y., et al.: Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. ASPLOS pp. 615–629 (2017)
4. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS (2012)
5. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: CVPR (2015)
6. Mao, J., et al.: MoDnn: Local distributed mobile computing system for deep neural network. In: DATE (2017)
7. Motamedi, M., Fong, D., Ghiasi, S.: Fast and energy-efficient CNN inference on IoT devices. arXiv preprint arXiv:1611.07151 (2016)
8. Redmon, J., Farhadi, A.: Yolo9000: Better, faster, stronger. arXiv preprint arXiv:1612.08242 (2016)
9. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
10. Teerapittayanon, S., McDanel, B., Kung, H.: Distributed deep neural networks over the cloud, the edge and end devices. In: ICDCS (2017)
11. Zhao, Z., Barijough, K.M., Gerstlauer, A.: DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. IEEE TCAD pp. 2348–2359 (2018)