International Journal of Parallel Programming (2021) 49:600–624
https://doi.org/10.1007/s10766-021-00712-3

DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices

Rafael Stahl[1] · Alexander Hoffman[1] · Daniel Mueller-Gritschneder[1] · Andreas Gerstlauer[2] · Ulf Schlichtmann[1]

Received: 20 April 2020 / Accepted: 24 March 2021 / Published online: 7 April 2021

Abstract

Performing inference of Convolutional Neural Networks (CNNs) on Internet of Things (IoT) edge devices ensures both privacy of input data and possible run time reductions when compared to a cloud solution. As most edge devices are memory- and compute-constrained, they cannot store and execute complex CNNs. Partitioning and distributing layer information across multiple edge devices to reduce the amount of computation and data on each device presents a solution to this problem. In this article, we propose DeeperThings, an approach that supports a full distribution of CNN inference tasks by partitioning fully-connected as well as both feature- and weight-intensive convolutional layers. Additionally, we jointly optimize memory, computation and communication demands. This is achieved using techniques to combine both feature and weight partitioning with a communication-aware layer fusion method, enabling holistic optimization across layers. For a given number of edge devices, the schemes are applied jointly using Integer Linear Programming (ILP) formulations to minimize data exchanged between devices, to optimize run times and to find the entire model's minimal memory footprint. Experimental results from a real-world hardware setup running four different CNN models confirm that the scheme is able to evenly balance the memory footprint between devices. For six devices on 100 Mbit/s connections the integration of layer fusion additionally leads to a reduction of communication demands by up to 28.8%. This results in run time speed-up of the inference task by up to 1.52x compared to layer 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.

Extended author information available on the last page of the article

1 Introduction

In the context of the Internet of Things (IoT), deep learning has emerged as a valuable tool. Being fed large, complex and noisy sensory input data sets, a deep learning model is trained offline with high computational effort to produce a set of cascading mathematical manipulations that transform new input data into an output classification result. The task of providing new input data to a pre-trained deep learning model, producing as output a classification, is called *inference* and is the focus of this article.

For image classification, Convolutional Neural Networks (CNNs) are a widely used deep learning architecture, given their ability to extract many complex high-level features needed for object classification. CNN inference is a very resource-intensive task given the large amount of input, model and intermediate data that must be stored and processed. IoT edge devices are components that operate at the edge between an IoT application and the physical world. Considering their constrained resources, it is often not possible to perform CNN inference on a single edge device such as a smart camera. While this could be solved by deploying more powerful edge devices, the cost of such a solution is prohibitive and may not be appropriate for the target application. For many applications the overall system's performance can be greatly improved by distributing more computation to other IoT edge devices [7]. An orthogonal solution is therefore the utilization of multiple cooperative edge devices to carry out the CNN inference task in a distributed and cooperative fashion. In many existing IoT applications, a large number of edge devices are available and connected with each other via some local network, for example, a cluster of surveillance cameras. This means that many existing IoT installations already have the required system architecture for performing distributed inference. Another advantage of distributed inference is that when inputs arrive, most devices are idle, given the low duty cycles of common IoT-enabled sensors. Thus, a low-cost but efficient solution can be established by utilizing the idle time of other edge devices.

We base our work on the prior DeepThings approach described in [29], which introduced an approach for memory- and communication-aware partitioning and fusing of feature-dominated convolutional layers. In [25], we previously extended DeepThings with methods to partition and fuse convolutional and fully-connected layers whose weight data size dominates their input and output data size. However, we only evaluated our approach in simulation for weight-dominated layers and did not account for all joint optimization opportunities. In this article, we describe DeeperThings, a comprehensive approach for the distributed execution of complete CNNs considering all layer-types while simultaneously optimizing for computation, memory and communication demands. The computation and memory footprint of processing and storing feature and weight data is evenly distributed over all devices, such that the CNN inference task can be scaled down for resource-constrained IoT edge devices. Additionally, the communication demand is minimized by DeeperThings by finding an optimized partitioning of the CNN. This article makes the following new contributions to enable the DeeperThings approach:

1. An improved description of the memory- and communication-aware partitioning and layer fusing scheme for fully-connected and weight-intensive convolutional layers. When combined with DeepThings' prior schemes on feature-intensive convolutional layer partitioning, DeeperThings enables complete distribution of arbitrary state-of-the-art CNNs.
2. A new Integer Linear Programming (ILP) approach to minimize the memory footprint of a full CNN model by finding the optimal point to switch from feature partitioning to weight partitioning.
3. An improved ILP approach to minimize the communication overhead by finding the Optimized Weight Partitioning (OWP) for all weight-partitioned CNN layers. A similar ILP approach was outlined in [25], but it did not consider the full potential to save communication overhead, which we now exploit with DeeperThings in this article.
4. We deploy and evaluate DeeperThings on a real-world IoT edge setup performing CNN inference. We only evaluated our prior approach from [25] in simulation on a single model. In this article, we perform a case study using four CNN models on a Raspberry Pi cluster to explore the trade-offs between run time, memory requirements and communication overhead for different network bandwidths and device counts. The new models were also used for extended evaluation of the two partitioning optimization methods.

Experimental results for DeeperThings show that the memory footprint scales down proportionally to the number of available edge devices. Optimizing the partitioning schemes with layer fusion leads to an up to 28.8% reduced communication demand, while executing the inference task on six edge devices with 100 Mbit/s connections for four evaluated CNNs: YOLOv2, AlexNet, VGG-16 and a GoogLeNet derivative. This results in a run time speed-up by up to 1.52x compared to a straight-forward layer partitioning. In the evaluation we additionally varied the available bandwidth and number of devices. The ILP optimization methods were evaluated using the same models. Compared to the hand-picked configuration of YOLOv2 in [29], the per-device memory footprint could be further reduced by 25%.

2 Related Work

Fully distributed inference is also tackled by the approach of MoDNN [16]. MoDNN distributes a CNN across multiple mobile phones connected via a wireless network. The approach also distributes both the layers' input and output data as well as the weight data across devices. While the approach is able to partition weights, it focuses mainly on sparse fully-connected layers (i.e., fully-connected structures, where some weights are zero). The approach does not take the communication between fully-connected layers into account, and weight-intensive convolutional layers are not addressed. Furthermore, the approach proposes to process networks in a layer-by-layer fashion, requiring all devices to synchronize by exchanging data after each layer. It would be possible to combine their method with the layer fusion and optimization methods proposed in this work. With additional constraints, the

communication overhead could be reduced further by the approaches presented in this work. Another full distribution is achieved with layer pipelining [18]. However, this method is not able to evenly distribute the memory demand for typical models.

Other works have addressed the execution of inference on edge devices with respect to pruning and quantizing the model, e.g., [3, 4, 17, 28]. Another approach is to design a new model architecture specifically for constrained devices [9]. A common solution is to offload inference to powerful servers in the cloud. However, this approach introduces other issues, such as input data privacy concerns and the requirement for a high bandwidth connection to the cloud [11, 23]. A load-aware approach presented in [27] focuses on partitioning and distributing parts of a model across different tiers of processing power. During inference, the model can stop at an intermediate layer if it has high confidence of a result. Therefore, for certain inputs, such a model would execute edge-only. Both of these works are orthogonal methods to the one proposed in this article. There exist various device-local methods to optimize performance and memory usage on a single device such as shrinking and pruning the CNN. Applying these methods can reduce output accuracy, thus no longer making the model a viable solution. As such there will always be models that are just too complex for a single device [4, 8, 14, 17].

With model distribution being an actively researched field, [1] presented work applicable to the distribution of CNN models targeting hardware accelerators. The central idea is that the fusion of the first few layers in the network reduces the total data transfer to and from the chip. Contrasting our work, the fusion method in [1] targets memory-constrained accelerators instead of similarly-constrained IoT edge devices. Fusing optimization for the accelerator is only investigated for the feature-intensive layers while the fusing approach presented in this article additionally targets the weight-intensive layers. Other works on accelerators have been focused on aggressive parallelization [2].

Other works focus on methods how the tasks within a network of collaborative edge devices should be distributed [6, 22]. However, these works handle general tasks and in contrast to our work focus on the network parameters. Our work deals with internals of fully-connected and convolutional operations to remove dependencies between such tasks which would have to be respected by more general approaches. Using larger scale edge devices for sharing work was explored in [12], but this has the disadvantage that a more powerful device is added to the network along with its additional power requirements.

The work in this article builds upon the previous DeepThings approach from [29], which addressed adaptive, distributed deep learning inference for systems with a dynamic availability of edge nodes. A fusing approach for multiple layers, which focuses on data partitioning in the feature-intensive layers, was the main contribution of this work. However, it did not consider weight partitioning. This is an important consideration because, given a sufficiently deep CNN, it is no longer possible for the presented approach to store the large volume of weight data on a single resource-constrained device after a certain layer-depth. DeepThings requires that weight-intensive network layers are evaluated on a central powerful gateway edge device that has sufficient memory. This constraint is removed by the DeeperThings approach presented in this work, enabling fully distributed inference

on a set of memory-constrained edge devices. Additionally, DeeperThings includes an ILP approach to minimize the communication demand by finding an optimized partitioning and fusing decision based on the CNN layer structure. The memory footprint per device is also optimized over the whole model by optimizing the layer at which to start weight partitioning.

3 Background on CNNs

The input of an image classification CNN, such as YOLOv2 [21], is an image where each color-channel of each pixel is represented by a neuron. The CNN input is therefore a three-dimensional structure, also called *tensor*, where the third dimension is a color intensity vector comprised of the three channels: red, green and blue. Input images are processed by a number of convolutional layers that apply multiple filter functions to their layer input to produce a set of feature maps. Feature maps give representations of how distinguishable certain features are within a given image. The sought-after features' characteristics are represented within the given filter functions. Thus, feature maps give accurate representations of the different characteristics of an image. With a high number of filters in each layer, the number of feature maps typically grows with each convolutional layer. Aside from convolutional layers there are also pooling layers, which shrink the width and height of feature maps in order to reduce the total size of the stored data, allowing a higher number of features to be extracted given a fixed data size. Thus, as the number of intermediate feature maps, and the respective number of characterizable features increases, each feature map loses resolution because of pooling. Because of this process, the input and output data dominate the memory usage for the first layers of a CNN, while in the later layers the weights dominate. The demands of weight data in later layers have made such layers the focus of this work, especially as distribution strategies for early layers were described in [29].

If a CNN is given the task of classifying what it sees, the last few layers typically consist of fully-connected layers, as seen in AlexNet [13] and VGGNet [24]. If instead the CNN's task is object localization within an image or classification of multiple objects, the later layers will also be convolutional. Such network architectures are known as Fully Convolutional Networks (FCNs) [15]. YOLOv2 is an FCN used in the evaluation of our proposed layer fusion method and is following the typical data size distribution structure outlined above.

3.1 Fully-Connected Layers

Before introducing our notation for CNNs, we start with fully-connected layers, which are a simpler layer type that is the main building block of artificial neural networks. A fully-connected layer has the distinctive property that all input neurons are connected to all output neurons. The fully-connected layer operation can be expressed as:

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

where for the l-th layer, $a_{l,m}$ is the m-th element of the input neurons vector $\mathbf{a}_l \in \mathbb{R}^{M_l}$, $b_{l,k}$ is the k-th element of the output neurons vector $\mathbf{b}_l \in \mathbb{R}^{K_l}$, $w_{l,m,k}$ 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 is the dominating factor in determining the memory demand of the inference task. Biases can also be considered weights, but since they are comparably negligible in size, they are not included in the notation of this work. The computation time is dominated by the number of multiplication operations R_l. The multiplications are the ones seen in Eq. (1). Thus, for a fully-connected layer l with the dimensions K_l and M_l, we obtain: $Q_l = R_l = K_l \cdot M_l$.

We use a special way to illustrate the layers in this article, which is shown in the example in Fig. 1. The given fully-connected layer has four input neurons, which are represented by the white nodes $a_{1,m}$. The gray nodes a_m represent the internal temporary nodes used for the fully-connected operation represented by the box. The arrow between the white and gray nodes show the flow of data. If the white input neurons are available on the device that is executing the operation, then this arrow requires no computational effort as the buffer of input neurons can be directly used by the operation. If the operation is executed on another device, the buffer needs to be communicated (later shown as dashed line). The w above many crossing arrows followed by summation boxes signifies the use of a dot product across an input vector and corresponding weight vectors. A box labelled with an f represents the application of an activation function. The gray nodes b_k represent the output neurons from the operation that become the input neurons $a_{2,m}$ for the next layer.

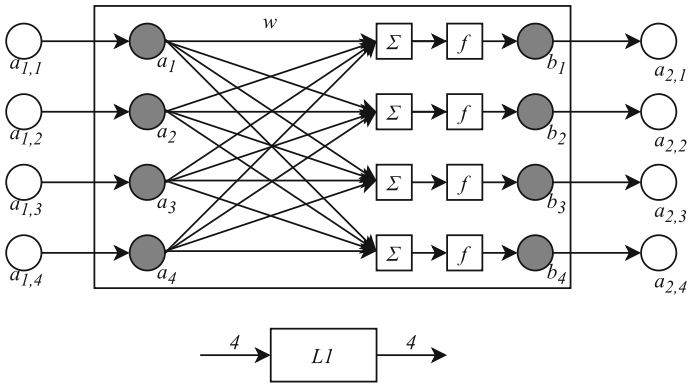


Fig. 1 Fully-connected layer

3.2 Convolutional Layers

A convolutional layer takes multiple two-dimensional feature maps as input, where each feature map captures one channel of the input tensor. For example, an RGB input image is represented as three feature maps, one map for each color channel. Taking all input feature maps into consideration, classical image filtering is applied on each input channel using a two-dimensional kernel. The results of this filtering are summed to create a single output feature map for all channels. As a convolutional layer usually applies multiple filters, the output of a convolutional layer respectively outputs multiple feature maps.

As such, the layer's operation can be expressed as:

$$\mathbf{B}_{o,l} = f\Big(\sum_{c=1}^{C_l} \operatorname{corr}(\mathbf{A}_{c,l}, \mathbf{W}_{c,o,l})\Big), \quad o \in \{1, \ldots, O_l\} \tag{2}$$

where the matrix $\mathbf{A}_{c,l}$ is the c-th 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 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 feature map of the input with the o-th feature map of the output. The kernels are contained in the four-dimensional weight tensor $\mathbf{W}_l \in \mathbb{R}^{U_l \times V_l \times C_l \times O_l}$, where U_l and V_l are the kernels' width and height, respectively. 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:

$$\operatorname{corr}(\mathbf{A}, \mathbf{W})_{x,y} = \sum_{(u=-\lfloor \frac{U}{2} \rfloor)}^{\lfloor \frac{U}{2} \rfloor} \sum_{(v=-\lfloor \frac{V}{2} \rfloor)}^{\lfloor \frac{V}{2} \rfloor} a_{x+u,y+v} w_{u,v} \tag{3}$$

Note that these formulas assume a stride of one and an input padding according to the kernel size. However, the methods described in this article still apply for strides larger than one. For convolutional layers the input size M_l, the output size K_l, number of weights to be stored Q_l and the number of multiplication operations R_l are computed as:

$$\begin{aligned} M_l &= X_l \cdot Y_l \cdot C_l, \qquad K_l = X_l \cdot Y_l \cdot O_l \\ Q_l &= U_l \cdot V_l \cdot C_l \cdot O_l, \quad R_l = X_l \cdot Y_l \cdot U_l \cdot V_l \cdot C_l \cdot O_l \end{aligned} \tag{4}$$

Due to their high dimensionality, convolutional layers of CNNs are thus computationally and memory intensive. Given resource-constrained edge devices, such layers must be distributed to make the CNN's execution possible.

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 requirements. Hence, we approximate the required memory footprint F_n as the number of weights to be stored on the n-th device. The

computational load results predominately from the multiplications performed using these weights. When distributing inference across multiple devices, these multiplications can be parallelized. The longest execution path, given by the number of parallelized multiplications, is denoted as T.

By parallelizing the workload, a synchronizing communication load is created. This load, denoted C with the unit *number of neurons*, is the result of exchanging input and output data between the devices. The value of C of course is only an estimation for the exact communication impact on a real CNN inference implementation. Nonetheless, our experimental results show that C is highly correlated to the run time of the inference task. As such, the communication overhead and the parallelization factor impact the overall run time in opposite fashions as most edge devices are bandwidth-constrained. Therefore, utilizing a larger number of devices leads to a lower per-device memory footprint F_n in exchange for larger communication overheads C.

Thus, considering a network with L layers mapped to a single device, we would obtain the following:

$$F_n^{(N)} = \sum_{l=1}^{L} Q_l, \quad T^{(N)} = \sum_{l=1}^{L} R_l, \quad C^{(N)} = 0 \tag{5}$$

This is illustrated in Fig. 2 for a fully-connected four-layer example. Edges are annotated with the input/output sizes of the respective layers. Thus from Fig. 2 follows: $F_1^{(N)} = 4 \cdot 8 + 8 \cdot 16 + 16 \cdot 4 + 4 \cdot 4 = 240$, $T^{(N)} = 240$, $C^{(N)} = 0$.

Layer pipelining as in [18] can be applied to the same example to distribute the layers across two devices, reducing F_n. We can map layers 1 and 2 to Device 1 and layers 3 and 4 to Device 2, as shown in Fig. 3. Intuitively, we obtain: $F_1^{(DL)} = 4 \cdot 8 + 8 \cdot 16 = 160$, $F_2^{(DL)} = 16 \cdot 4 + 4 \cdot 4 = 80$, $T^{(DL)} = 240$ and $C^{(DL)} = 16$. The operational memory footprint is determined by the larger set of layers, in this case $F_1^{(DL)}$. This is 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, now with additional communication overhead. This mapping can be improved by using layer partitioning, which will be the focus of the remainder of the article.

4 Layer Partitioning Methods

Depending on the size of input/output data (features) or weights, the layer data should be partitioned by either features or weights to achieve a reduced memory footprint per device. The following two sections describe these methods in more detail.

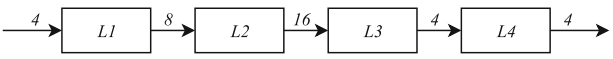


Fig. 2 4-Layer example on a single device

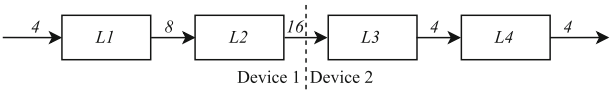


Fig. 3 4-Layer example with sequential layer mapping

4.1 Partitioning of Feature-dominated Layers

Partitioning the features of CNN layers is useful when their size dominates a layer's memory usage. In typical CNN model structures, this is the case for early layers, because their feature resolution is large and there are fewer convolutional filters. To support our holistic partitioning of CNNs, feature partitioning must also be supported. The state-of-the-art method presented in [29] is included in our solution for this purpose. In that work, an approach called Fused Tile Partitioning (FTP) is presented that first partitions the input and output feature maps of multiple layers into sets of tiles in an *N*x*N* grid. It then fuses corresponding tiles across layers to exploit inherent locality in convolutions. This results in partitions with a set of *N*x*N* fused tile stacks that can be executed independently. Since the computation of each consecutive layer only depends on the previous layer's respective tile, this connection (or fusion) does not require synchronization between devices. The method achieves reduced memory footprint from the feature map division and reduced communication overhead because synchronization is not required. FTP is only applicable to convolutional layers and not fully-connected ones, because it exploits the structure of the convolution operation. Due to the nature of fully-connected layers, they are however always weight-dominated.

4.2 Partitioning of Weight-dominated Layers

The core mechanism of our approach is the application of partitioning to weight-dominated layers of a CNN such that weight data and computational load is evenly distributed across all available devices, whilst minimizing the inference run time. In the following, we first discuss partitioning schemes with respect to fully-connected layers. In [25], we presented a novel approach for communication-aware partitioning of such weight-dominated fully-connected layers. A weight partitioning scheme can be achieved by partitioning weights such that either inputs or outputs of a layer are split and mapping one partition of each layer to a respective device. Weight partitioning by splitting input and output data is simple and very effective when distributing weight data whilst minimizing communication overhead. In [25] we also discussed how partitioning schemes for fully-connected layers can be further improved in terms of communication overhead by applying a newly proposed layer fusing scheme. Finally, we extended all partitioning and fusing schemes to weight-intensive convolutional layers. In this article, we present a more precise description of the layer communication demands, which considers that the schemes can be used jointly such that they are able to reuse layer data maximally.

4.2.1 Layer Output Partitioning (LOP)

Layer Output Partitioning (LOP), as illustrated in Fig. 4a, uses all of the inputs $\mathbf{a}_l$ and a partition of $\mathbf{W}_l$ to calculate a subset of the output neurons $\mathbf{b}_l$. The output subset can then be finalized by applying the activation function f. The final partitioned output values must then only be concatenated (*Concat/L1C*) to obtain the full output vector $\mathbf{b}_l$, thus synchronizing the layer's output across all devices. Partition 1 and partition 2 can be executed in parallel by different devices. The dashed arrows in the figure represent data flow that is subject to inter-device communication.

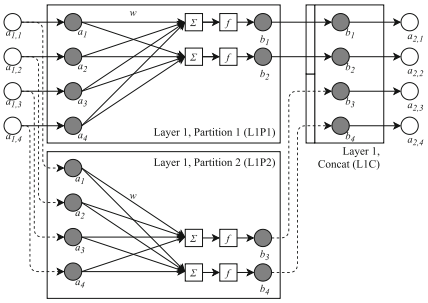
Assuming LOP is used for all L layers, given N devices, we obtain the memory footprint $F_n^{(LOP)}$, the execution time $T^{(LOP)}$ and the communication demand $C^{(LOP)}$ as follows:

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

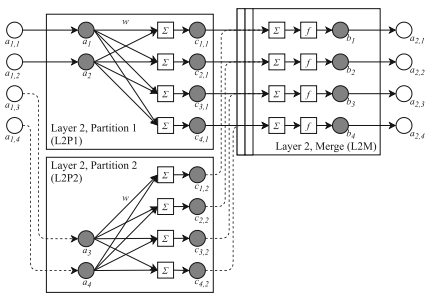
Independent of the remaining network structure, for any layer using LOP, we obtain:

$$C_l^{(LOP)} = (1 - r_{l-1})M_l(N-1) + r_l K_l (N-1) + (1 - r_l)K_l \frac{N-1}{N} \tag{7}$$

The Boolean variable r_l is 1 if data reuse between layers l and $l+1$ is possible. This is the case when an LOP layer is followed by another LOP layer. This partial communication intuitively means that part of the output is already ready in memory for the next layer, thus part of the output's data can be reused without the need for additional synchronization. This is seen in Fig. 5 through the communication arrows that do not cross the inter-device boundary. The first summand in Eq. (7) counts the input distribution to other $(N-1)$ devices, only if there is data reuse between the current and previous layer. This can be seen in Fig. 5 where the first LOP operation requires the distribution of the 4 inputs, but all following operations can reuse the



(a) Layer Output Partitioning (LOP).



(b) Layer Input Partitioning (LIP).

Fig. 4 Layer partitioning methods

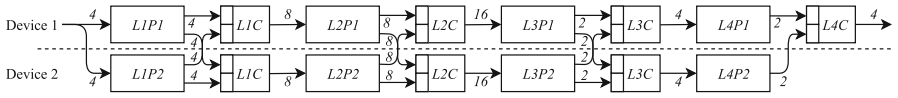


Fig. 5 4-Layer example with Layer Output Partitioning

data out of the *Concat* blocks. The second and third summands in Eq. (7) describe the output distribution to the *Concat* blocks. If there is no reuse between the current and next layer, the partial data only needs to be sent back to the initiating device for concatenation as seen for the last LOP operation. Otherwise, the partial data needs to be shared with all other devices, such that they can concatenate locally. The communication arrows that do not cross the inter-device boundary mean that the data is already ready in memory and there is no need for additional synchronization.

Compared to the description in [25] that only refers to a partitioning that uses LOP on all layers, Eq. (7) is now applicable for the layer partitioning type optimization presented in Section 5.2, because it describes reuse of previous layer data more generally instead of only for LOP.

Applying LOP to all layers of our example as shown in Fig. 5, 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. The memory footprint is thus evenly balanced allowing for full utilization of parallelism, whilst requiring some inter-device communication.

4.2.2 Layer Input Partitioning (LIP)

The second method, shown in Fig. 4b, is Layer Input Partitioning (LIP). LIP takes a partitioned subset of $\mathbf{a}_l$ to calculate a respective partial output of $\mathbf{b}_l$. While we are now using a different subset of weights, the number of weights required per device remains unchanged compared to LOP. As the output values are only an incomplete part of the complete output nodes, they must be summed before being passed to the activation function f. As such, f can not be executed within the partition and a merge operation is required that sums the output values before activating the layer.

If LIP was used on all l layers for all N devices, the same memory footprint, $F_n^{(LIP)} = F_n^{(LOP)}$, would be obtained. The same execution time, $T^{(LIP)} = T^{(LOP)}$, is obtained as when applying LOP. The respective communication demand using LIP is:

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

Using LIP, the complete set of outputs for each device must be communicated to the device performing the merging, while the inputs into the LIP operation only have to be partly communicated, providing no opportunity for data reuse. This results in large communication overhead when only LIP is used. Applying LIP to the aforementioned example would result in a communication overhead of $C^{(LIP)} = 48$. However, when the number of inputs to a layer is significantly larger than the

number of outputs, applying LIP to that particular layer selectively can still reduce the overall communication demand over other layer partitioning types like LOP.

While other weight partitioning schemes are possible, they cannot have a lower total input or output size compared to strict LOP or LIP partitioning, because they can only be less or equally balanced. The methods are compatible with sparse weight data, since for those, only the non-zero weight data have to be distributed evenly.

4.2.3 Fused Layer Partitioning (FUSE)

As *Concat* and *Merge* operations require input from all partitioned data subsets, they are considered synchronization operations. Synchronization operations carry a large communication overhead. As such, optimizations to overall execution time can be made by performing more optimal synchronization placement. When combining both LOP and LIP, we are able to *fuse* layers in a way that one such synchronization point is eliminated, as seen in Fig. 6. The intermediate layer neurons stay local to the devices and do not contribute to any communication demands.

By applying LOP to the input of the combined operation's first layer, the interim data now has the appropriate shape to be passed as the input to a LIP operation, applied to the following network layer. The resulting output of the combined operation has the correct shape for synchronization. Synchronization is inevitable at this point, because the output only represents partial values of the distributed layer's output, a property inherent of LIP operations. As such, these partial output values

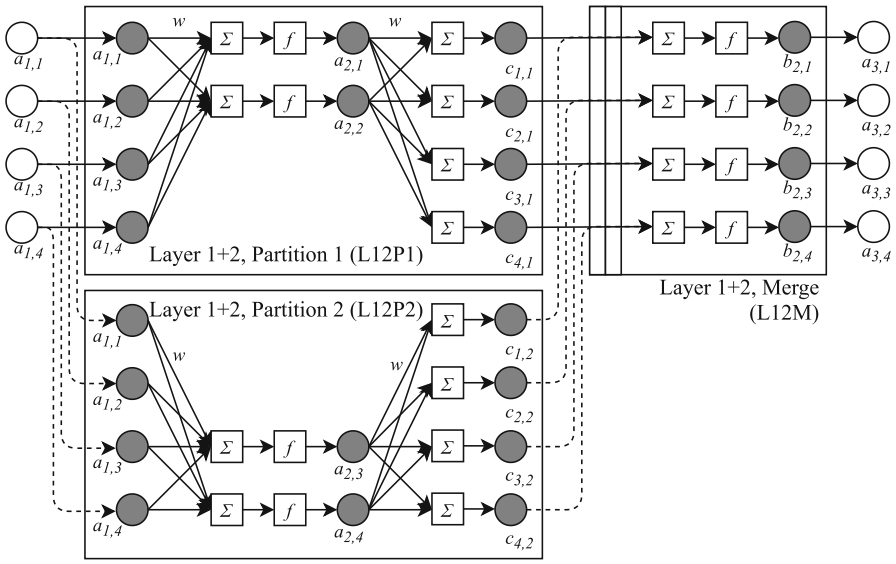


Fig. 6 Fused Layer Partitioning (FUSE)

must be merged for summation and activation. This implies that only two consecutive layers can be fused using this method before requiring synchronization. Given the benefits of fusing layers, it should be noted that not all layers of the network need to be fused. LOP and LIP operations can still be applied individually to reduce the size of required memory, naturally at the expense of communication overhead. For example, networks with odd layer counts cannot have all network layers fused. For fused operations we obtain the same memory footprint and execution time.

$$F_n^{(FUSE)} = F_n^{(LOP)} = F_n^{(LIP)}, \quad T^{(FUSE)} = T^{(LOP)} \tag{9}$$

While the execution time does not increase, the overall run time of using fused operations, similarly to using individual LIP and LOP operations, is increased due to the incurred communication overhead.

Even though a fusion on two layers could be described as a single new layer, they will still be described in terms of the two original layer indices, to more easily compare different fusing configurations. The communication demands C for the first and second part of the fused layers are described as:

$$C_l^{(FUSE1)} = (1 - r_{l-1})M_l(N-1), \quad C_l^{(FUSE2)} = K_l(N-1) \tag{10}$$

The first fused layer is equivalent to LOP without the outputs and the second fused layer is equivalent to LIP without the inputs as these do not need to be communicated due to fusing. When multiple devices are executing their fused partition, they work in parallel on their share of the first and second part of the fused layers. Note that if data reuse is possible between the first fused layer and the previous one, there is no communication required for the first fused layer. As outlined before, this description of $C_l^{(FUSE1)}$ is improved over [25], because it now considers data reuse from a possible previous LOP layer.

When applying Fused Layer Partitioning with two fusions to our example as shown in Fig. 7, we observe: $C = 4 + 16 + 16 + 4 = 40$ while F and T remain unchanged compared to LOP. The observed result is worse than applying LOP operations to every layer. This is because two fusions are not a good solution for the given example. This demonstrates that fusion decisions need to be taken judiciously in order to achieve improvements in execution, as will be further discussed in Section 5.

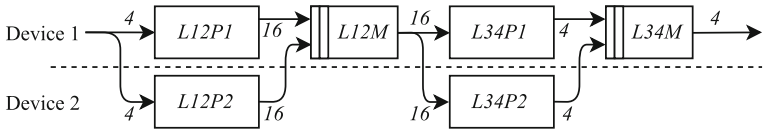
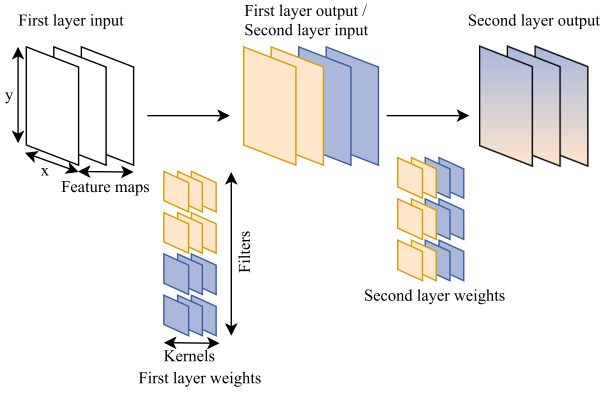


Fig. 7 4-Layer example with Fused Layer Partitioning

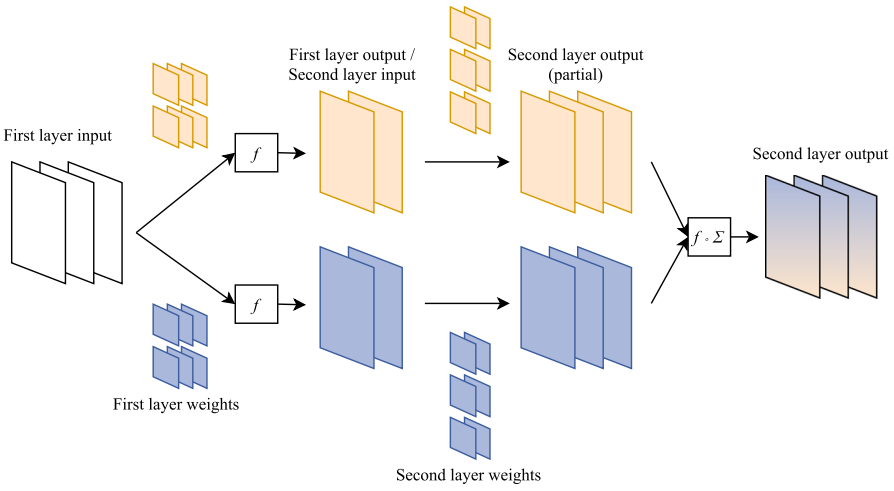
4.2.4 Partitioning of Convolutional Layers

The proposed layer partitioning and fusing schemes can also be applied to convolutional layers as illustrated in Fig. 8. In that figure, the weights have been partitioned into two equal partitions highlighted in two different colors. Compared to fully-connected layers, the output feature maps of the first layer are partitioned instead of the output neurons.

A convolutional layer extracts n feature maps from n filters with each filter being comprised of a number of kernels. The number of kernels required in every filter is



(a) Before partitioning.



(b) After partitioning.

Fig. 8 Weight partitioning for two convolutional layers

determined by the number of channels of the input to the convolutional layer. For example, an RGB image is comprised of three channels with each pixel being the combination of the three. An RGB image would require filters with three kernels, one per channel. The application of a kernel then generates an interim per-channel output that is then summed to produce the output of the filter. As such, a convolution on an RGB image would apply a filter, comprised of three kernels, to produce a single-channeled output that is the superposition of each kernel's output from its respective channel.

Partitioning of two convolutional layers means that the filters of the first layer are partitioned between devices. The first layer on each device thus outputs a single channel of the input to the second layer, as can be seen in Fig. 8b. The second layer is partitioned such that the kernels within the filters are partitioned across devices. Therefore, each device does not need to contain all kernels for each filter.

Partitioning the first layer only applies a subset of the network's filters, in turn producing a subset of the intermediate feature maps (channels) used as input into the second convolutional layer. Mathematically, this is equivalent to implementing Eq. (2) only for a subset of the o indices assigned to the device, meaning the device only needs to store its own filter weights. In the second layer, all of the filters, with a subset of kernels, are applied to each of the input feature maps, such that each device calculates its portion of the to-be-merged output. As such, only the kernels (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. (2), but now only for a subset of the c indices.

Similarly to layer fusion for fully-connected layers, non-linear activation functions cannot be performed before the complete output set is complied, as c is only a subset of the output. Thus, a merge step is again required to sum up the partial results, allowing for the application of the activation function, thus completing the layer. Characterizing the fusion of two convolutional layers is done using the same formulations for F_n, T and C, given in Equations (9) and (10), where the layer properties M_l, K_l, Q_l and R_l must be taken from Eq. (4) for convolutional layers. Similarly to 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 when using distributed inference. This is especially useful as the memory footprint required to store the weight tensors for later convolutional layers of CNNs is often very high.

5 ILP-based Optimization of Partitioning Decisions

There exist several degrees of freedom to partition each CNN for fully distributed inference. We propose a two-step process solving two ILPs. The first ILP optimizes the memory footprint per device by deciding when to use feature partitioning versus weight partitioning. The second ILP optimizes the communication demand in the weight partitioned CNN layers by selecting which method of weight partitioning to use.

5.1 ILP-based Memory Footprint Minimization

For standard CNNs, the earlier layers are input/output dominated and should run on FTP presented in [29], while the later layers are usually weight dominated layers, which should use weight partitioning. To reach an optimal memory footprint per device, we use the following ILP optimization to identify the point at which to switch from FTP to weight partitioning:

$$min_{\mathbf{a},\mathbf{b},c}\ F_n^{(FULL)} = c + \sum_{l=1}^{L} \left(b_l \frac{Q_l}{N} + (1 - b_l) Q_l \right) \tag{11}$$

$$s.t.\ \forall_{l=1\ldots L} \quad a_l = b_l (M_l + K_l) + (1 - b_l) \frac{M_l + K_l}{N} \tag{12}$$

$$\forall_{l=2\ldots L} \quad b_l \geq b_{l-1} \quad b_l \in \{0, 1\} \tag{13}$$

$$\forall_{l=1\ldots L} \quad c \geq a_l \quad c, a_l \in \mathbb{N}, \tag{14}$$

where a_l, described by (12), are integer variables that hold the memory footprint attributed to the inputs and outputs of layer l. b_l are Boolean variables that are true when layer l is using weight partitioning and false if it is using feature partitioning. In weight-partitioned layers ($b_l = 1$), all input/output data has to be duplicated on each device, whereas in feature-partitioned layers ($b_l = 0$) it is partitioned by the number of devices. This formulation includes some small simplifications, since FTP has a "slightly larger (3%)" [29] memory footprint due to overlapped data. Moreover, LOP or LIP layers do not require the full input or output. Our experimental results will evaluate the actual methods without these simplifications, which are only present in this ILP formulation. (13) ensures that the layer partitioning can only switch once from feature partitioning to weight partitioning. c is an integer variable that represents the maximum of a_l, because only the highest input/output pair counts towards the total memory footprint. Since the objective function is minimized, it is sufficient to specify the constraint (14) to achieve the desired equivalency $c = max_l(a_l)$. The total memory footprint per device is described in (11) as the the sum of the maximum input/output data footprint and the sum of all layers' weight data footprint. In weight-partitioned layers ($b_l = 1$), the weight data is divided by the number of devices, whereas in feature-partitioned layers ($b_l = 0$) all weights have to be duplicated on each device.

5.2 ILP-Based Communication Optimization for Weight Partitioned Layers

In the later weight-dominated layers, fusing at every possible opportunity, as done in Fig. 7, can possibly be an inferior solution to the careful selection between LIP, LOP and fusing per layer. This is because communication is dependent on the output and input layer sizes, which can vary greatly between layers, and given that layers can only be fused pairwise. Fusion should be performed such that communication sizes between fused layer pairs are minimized. If a layer's output

or input is fused, its input or output can in turn no longer be fused and must be communicated. Additionally, a non-fused layer may favor either LOP or LIP partitioning schemes. Given a network of L layers, we define $o_l = 1$ if layer l is using LOP, similarly $i_l = 1$ if it is using LIP, $f_l = 1$ for the first part of a fused layer and $s_l = 1$ for the second part of a fused layer, otherwise all variables are zero. With this we can formulate the communication demand C for the OWP as:

$$C^{(OWP)} = \sum_{l=1}^{L} \left(o_l C_l^{(LOP)} + i_l C_l^{(LIP)} + f_l C_l^{(FUSE1)} + s_l C_l^{(FUSE2)} \right) \tag{15}$$

We propose that partitioning and fusion decisions need to be made such that the communication demand C is minimized. This is done with the following ILP optimization:

$$min_{\mathbf{o},\mathbf{i},\mathbf{f},\mathbf{s},\mathbf{r}} \; C^{(OWP)} \tag{16}$$

$$s.t. \; \forall_{l=1\ldots L} \quad o_l + i_l + f_l + s_l = 1 \quad o_l, i_l, f_l, s_l \in \{0,1\} \tag{17}$$

$$\forall_{l=1\ldots L-1} \quad s_{l+1} = f_l \tag{18}$$

$$\forall_{l=1\ldots L-1} \quad r_l = o_l(o_{l+1} + f_{l+1}) \quad r_l \in \{0,1\} \tag{19}$$

$$f_L = 0 \qquad s_1 = 0 \qquad r_L = 0 \tag{20}$$

Here, (17) assures that only one partitioning scheme is chosen per layer, (18) assures that the second fused layer is performed directly after the first fused layer and (20) prohibit illegal fusing pairs on the boundaries and disallow data reuse from the last layer. (19) defines that data reuse happens when the current layer is LOP and the one after that is either LOP or a first fused layer. (19) and some summands of Eq. 15 contain products of variables, but since they are binary variables, they can be linearized with a helper variable and additional constraints as follows [5]:

$$c = ab \quad a, b, c \in \{0,1\} \tag{21}$$

$$c \leq a \qquad c \leq b \qquad c \geq a + b - 1 \tag{22}$$

Optimizing the partitioning and fusion decisions for our running example leads to the optimized solution shown in Fig. 9. Layers 1 and 4 both employ LOP, while layers 2 and 3 are fused. The calculated communication overhead of $C^{(OWP)} = 22$ shows a significant reduction given the partitioning and fusion optimizations. The

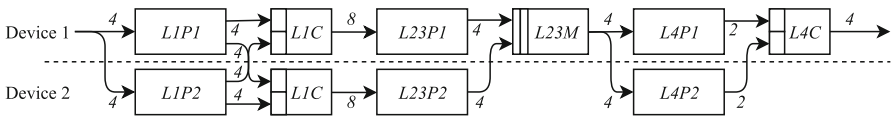


Fig. 9 4-Layer example with Optimized Weight Partitioning

previously most significant communication demand was present between layers 2 and 3, as seen in Fig. 7. This demand could be eliminated through the fusion decisions seen in Fig. 9. Summarizing, the ILP optimization considers the input and output sizes of all layers to decide on the best partitioning scheme of each layer.

6 Experimental Evaluation

We evaluated DeeperThings using the popular CNN models YOLOv2 [21][1], AlexNet [13][2], VGG-16 [24][2] and a GoogLeNet derivative (called "Extraction") [26][2]. Our optimization methods are applied to these models to find their optimal partitioning configuration. Then, the models are deployed on a Raspberry Pi 4 edge cluster for measurements of run time and memory usage.

6.1 Evaluation of the ILP-based Optimization Methods

The optimization methods to solve the ILPs in Eq. 11 and Eq. 16 were implemented using the "Coin-or branch and cut" ILP solver included in OR-Tools [10, 19]. Using an Intel Core i5-7500 machine running at 3.4 GHz, all evaluations completed in less than a second.

6.1.1 Evaluation of ILP-based Memory Footprint Minimization

Table 1 shows the different models with their number of layers L and the optimized layer at which to switch from feature to weight partitioning. It lists the memory footprint per device $F_n^{(FULL)}$ using a cluster of ten devices compared to a single device performing the full network inference with footprint $F_n^{(SINGLE)}$. The layer that was manually picked in [29] to switch away from feature-intensive partitioning was layer 17 for the YOLOv2 model. The choice was made manually based on analysis of the memory usage per layer. When evaluating Eq. (11), this results in $F_n^{(FULL)} = 37.9$ MB which is 33% higher than the automatically optimized solution that switches at layer 13 (which is equivalent to a 25% reduction). Using sequential layer mapping we can only reduce the memory footprint per device down to the largest memory requirement of a single layer, which is the best-case scenario. Additional layers possibly have to be mapped onto the device with the largest layer to limit communication demand. For our example networks this memory demand is shown in the last column as $F_n^{(SEQ)}$. Compared to sequential layer mapping we can arbitrarily increase the number of devices to reduce the memory footprint per device.

[1] https://pjreddie.com/darknet/yolov2 - YOLOv2 608x608

[2] https://pjreddie.com/darknet/imagenet/#pretrained - AlexNet, VGG-16, Extraction

Table 1 Memory footprint reduction for ten devices

Model	L	First OWP layer	$F_n^{(SINGLE)}$ [MB]	$F_n^{(FULL)}$ [MB]	F_n reduction	$F_n^{(SEQ)}$ [MB]
YOLOv2	32	13	256	28.4	9.0x	51.8
AlexNet	14	3	16.8	2.65	6.3x	5.82
VGG-16	25	8	84.5	13.2	6.4x	25.8
GoogLeNet	27	5	97.5	12.2	8.0x	20.1

6.1.2 Evaluation of ILP-based Communication Optimization

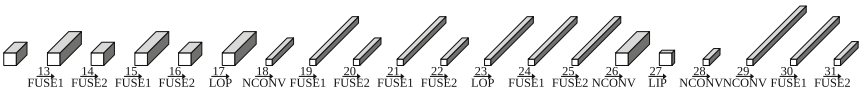
To evaluate the ILP optimization for finding an optimal weight partitioning (OWP), different CNN models were deployed to a system architecture with six edge devices.

The evaluation of the ILP on the different models is shown in Fig. 10. The number above the arrow is the layer index starting at the first weight-partitioned layer, the volumes represent layer input/output sizes. Below the arrows, the layer weight partitioning type chosen by the ILP optimization is shown or *NCONV* if the layer is not convolutional. Whenever a volume lies between *FUSE1* and *FUSE2* types, it does not contribute to the total communication demand because neurons do not have to be communicated between fused layers.

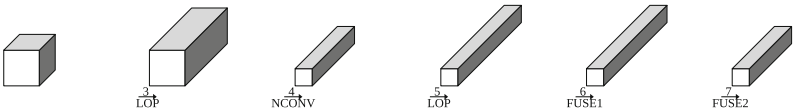
Table 2 contains the different models and compares the communication demand resulting from applying LOP at every layer ($C^{(LOP)}$) to the demand of OWP ($C^{(OWP)}$). Savings between about 6 to 29% indicate that the ILP can provide good decisions for the weight partitioning selection for CNNs. For comparison, the last column shows the solutions picked by the optimization in [25]. We updated the optimization to use improved estimate of C as evaluated by the formula in this article, which also considers data reuse. The new ILP presented in this article finds a better solution for the YOLOv2 and VGG-16 models because it additionally incorporates the LIP method.

6.2 Evaluation on Raspberry Pi Edge Cluster

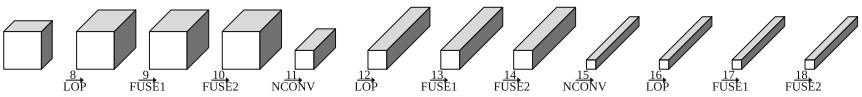
In [25], we evaluated our approach for weight partitioning on a Docker-based virtual device cluster emulated on an x86_64 desktop machine. In this article, we validated our complete DeeperThings approach on a physical edge cluster consisting of six Raspberry Pi 4 devices. They have a quad-core ARM Cortex-A72 processor, 2 GB of RAM and a gigabit Ethernet interface. Their operating system is *Raspberry Pi OS (32-bit) 2020-02-13* which includes the Linux Kernel *4.19.97*. We implemented our DeeperThings framework for fully distributed inference on top of the framework from [29], which incorporates the Fused Tile Partitioning for the earlier layers, by adding the presented OWP method for the later layers. The Deep(er)Things framework is based on the DarkNet neural network framework [20] with an added patch for use of the NNPACK acceleration library tuned for the ARM Neon SIMD instruction set extension. Along with the compiler



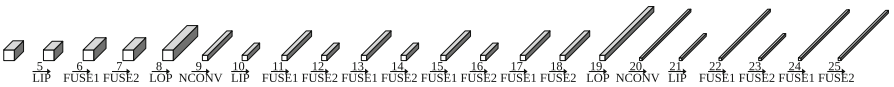
(a) OWP for YOLOv2.



(b) OWP for AlexNet.



(c) OWP for VGG-16.



(d) OWP for Extraction GoogLeNet derivative.

Fig. 10 Optimized Weight Partitioning for different CNNs

Table 2 Communication savings of OWP over LOP for six devices

Model	$C^{(LOP)}$ [MB]	$C^{(OWP)}$ [MB]	$C^{(OWP)}$ Saving [%]	C([25]) [MB]
YOLOv2	84.0	59.8	28.8	61.0
AlexNet	9.69	9.11	5.96	9.11
VGG-16	202	182	10.1	186
Extraction GoogLeNet	47.0	41.8	11.1	41.8

optimization flag *-Ofast* this ensures competitive inference performance. 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. This differs from the work in [29], where all processing of the weight-dominated later layers of the CNN is delegated to a single powerful central gateway device. We adapted the framework such that the source device collects the initial results from the earlier CNN layers preparing them for further distribution. The central gateway device is kept in our implementation for device discovery and coordination, but there is no technical limitation that would prevent such a network

to run purely on peer-to-peer technology. The prior existing communication pattern did not require long-running connections between devices for back and forth communication. However, with fusion of weight partitioned layers, all output data has to be synchronized at least every two layers between worker devices. As a result, a large number of messages must be exchanged between the source and worker devices. For the overall run time, it is therefore essential that the connections between the source device and all worker devices remain open, this has been 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. The run time is measured from start of inference until the final inference result has been calculated. The cluster consisted of six devices connected by a gigabit network switch. A seventh Raspberry Pi device acts as a gateway device, responsible for the coordination of intra-cluster communication. For each configuration, ten measurements were taken and the results were averaged to take run time variability into account. The memory measurements had negligible variation of one or two pages (4-8 kB), so only one measurement is reported. The Linux tool *tc* was used to impose software-based bandwidth throttles on the device's Ethernet interfaces, thus allowing for the simulation of bandwidth reductions. By being able to control the bandwidth linking the devices, we were able to magnify the communication overhead effects on inference partitioning, as will be seen when dealing with low bandwidths, such as 10 Mbit/s. Peak memory usage was measured with the Linux */proc/pid/status* file, which includes a value *VmPeak* to measure "Peak virtual memory size".

Fig. 11 shows measurements of DeeperThings peak memory usage savings over a single device. Due to limited development time, all weights are loaded before being pruned. Therefore, absolute memory usage is not shown since it had to be artificially increased to accurately measure peak usage. Measurements were taken before inference, because inefficient use of runtime queues skewed the peak usage after inference. However, a memory baseline can be taken from $F_n^{(SINGLE)}$ in Table 1. Current state-of-the-art edge inference frameworks such as *TensorFlow Lite for Microcontrollers* can achieve memory overheads below 50 kB[3]. When increasing the number of devices, we observe the expected proportional decrease of the memory footprint per device according to Eq. (11). This enables balanced inference scaling across the available memory resources of the edge devices equally for both LOP and OWP.

Fig. 12 shows the total inference run time speedup over a single device for different edge device and network parameters while using DeeperThings with all weight partitioned layers using LOP vs. OWP. The baseline run time values for one device are 13.4s ± 0.40s for YOLOv2, 0.98s ± 0.022s for AlexNet, 6.80s ± 0.25s for VGG-16 and 2.22s ± 0.022s for GoogLeNet. The results from the previous

[3] https://www.tensorflow.org/lite/microcontrollers

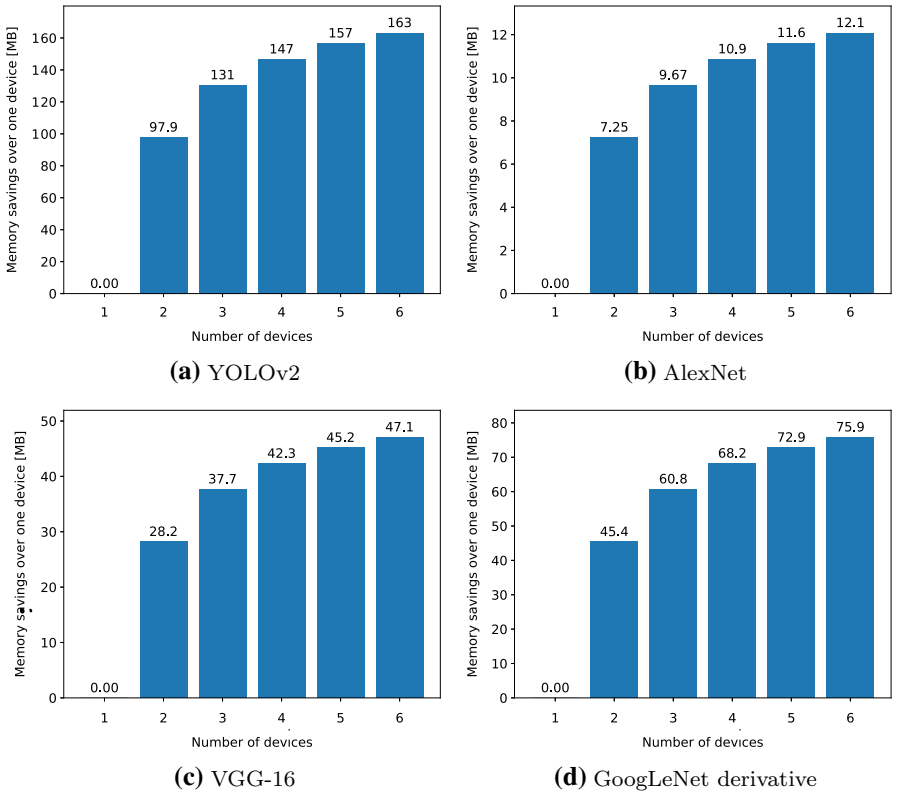


Fig. 11 Memory savings results

section were applied to decide at which point to switch from feature to weight partitioning and for finding the OWP. Note that the switching point was optimized for ten devices and kept the same for one to six devices, because this will keep the share of feature- and weight-partitioned layers constant to focus on the scaling of multiple devices. The measurements on the Raspberry Pi Cluster follow the pattern of the Docker setup used in [25], but showed lower variance. This is likely because on the Docker setup, all applications are competing for the same Linux scheduler and virtualized network, causing more interference. The CPU utilization limitation also adds additional variance, compared to non-throttled hardware. Existing work does not partition weight-dominated convolutional layers. Hence, we can only compare to a baseline being executed on a single node. When using idle nodes to distribute the inference task, a run time speed-up was observed. Note that for AlexNet the 1 Gbit/s results slow a general slowdown when using OWP. This is possible through measurement inaccuracies, because AlexNet has the lowest theoretical savings as shown in 2, and the high bandwidth makes the communication savings matter even less. There are several factors that influence the inference task's final run time when compared to the theoretical values for T and C given varying numbers of edge devices. Firstly, the effects of communication latency have to be

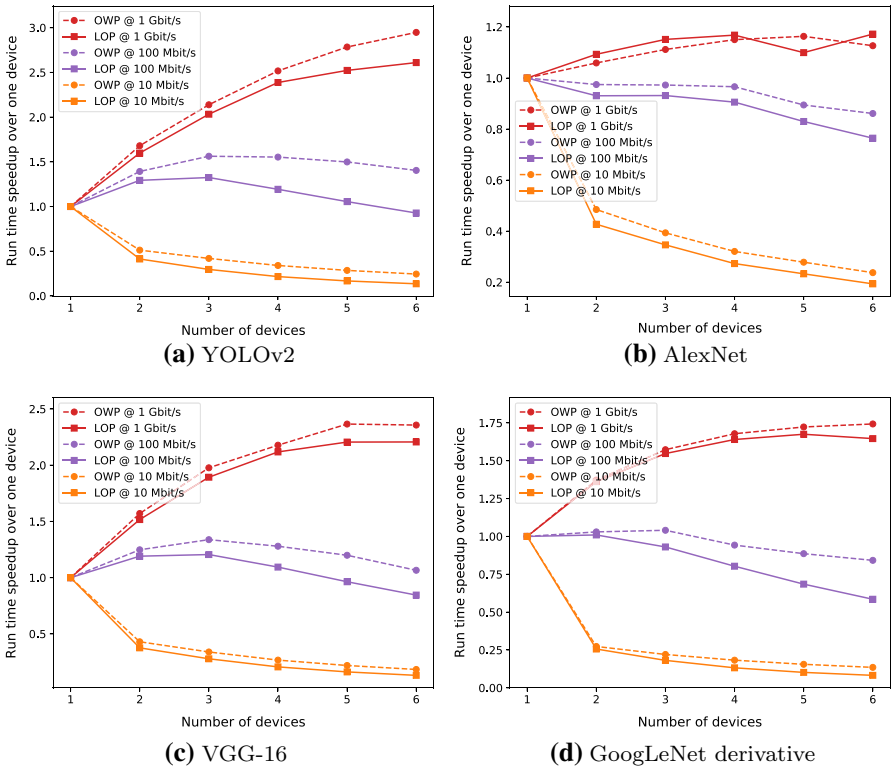


Fig. 12 Run time speedup of DeeperThings FTP+LOP vs. FTP+OWP

considered for each message between devices. Secondly, communication and parallelism have opposing impacts on run time. As such, when increasing the cluster size from three devices to six devices, no clear run time trend was observed. When comparing layer fusion to applying stand-alone LOP, a clear benefit can be seen in terms of run time speed-up. The benefit is less significant with very high available bandwidth (1 Gbit/s) as the Fusing only improves communication demand, which does not act as a bottleneck given such network speeds. The speed-up of OWP is generally higher the more devices are involved, because there is more communication happening which can positively be affected by layer fusion. However, as the cluster size increases, the total run time may also increase again, depending on bandwidth. Nonetheless, a significant speed-up was observed when using layer-fusion, which does not impose additional run time or memory costs.

7 Summary and Conclusions

In this article, an approach targeting memory-constrained edge devices was presented for partitioning and fusing of both feature- and weight-intensive fully-connected and convolutional CNN layers. The proposed method allows for

complete distributed execution of a CNN application across a cluster of resource-constrained edge-devices. As a result, communication overhead is reduced, resulting in run time speed-up. Additionally, ILP optimization methods are given, allowing for optimal partitioning and fusing decisions to be made given a preexisting CNN.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarartions

Conflicts of interest The authors declare that they have no conflict of interest.

References

1. Alwani, M., Chen, H., Ferdman, M., Milder, P.: Fused-layer CNN accelerators. In: IEEE/ACM International Symposium on Microarchitecture (2016)
2. Arredondo-Velázquez, M., et al.: A streaming architecture for convolutional neural networks based on layer operations chaining. J. Real Time Image Process. (2020)
3. Ayinde, B.O., Inanc, T., Zurada, J.M.: Redundant feature pruning for accelerated inference in deep neural networks. Neural Netw. **118**, 148–158 (2019)
4. Bhattacharya, S., Lane, ND.: Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: ACM Conference on Embedded Network Sensor Systems (2016)
5. Bisschop, J.: AIMMS optimization modeling. Lulu. com (2006)
6. Chen, J., et al.: iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks. IEEE Internet Things J. **6**(4), 7011–7024 (2019)
7. Chien, SY., et al.: Distributed computing in IoT: System-on-a-chip for smart cameras as an example. In: Asia and South Pacific Design Automation Conference, IEEE (2015)
8. Huynh, LN., Balan, RK., Lee, Y.: Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In: Workshop on Wearable Systems and Applications, ACM (2016)
9. Iandola, FN., et al.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. arXiv preprint arXiv:160207360 (2016)
10. johnjforrest, et al.: coin-or/cbc: Version 2.10.5. (2020) https://doi.org/10.5281/zenodo.3700700
11. Kang, Y., et al.: Neurosurgeon: collaborative intelligence between the cloud and mobile edge. ACM SIGARCH Comput. Arch. News **45**(1), 615–629 (2017)
12. Khelifi, H., et al.: Neurosurgeon: ccollaborative intelligence between the cloud and mobile edge. IEEE Commun. Lett. **23**(1), 615–629 (2018)
13. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Commun. ACM **60**(6), 1097–1105 (2017)
14. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: IEEE Conference on Computer Vision and Pattern Recognition (2016)
15. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: IEEE Conference on Computer Vision and Pattern Recognition (2015)

16. Mao, J., et al.: MoDnn: Local distributed mobile computing system for deep neural network. In: Design, Automation & Test in Europe, IEEE (2017)
17. Motamedi, M., Fong, D., Ghiasi, S.: Fast and energy-efficient CNN inference on IoT devices. arXiv preprint arXiv:161107151 (2016)
18. Martins Campos de Oliveira, F., Borin, E.: Partitioning convolutional neural networks to maximize the inference rate on constrained iot devices. Future Internet **11**(10), 209 (2019)
19. Perron, L., Furnon, V.: Or-tools. (2019). https://developers.google.com/optimization/
20. Redmon, J.: Darknet: open source neural networks in c. (2013–2016). http://pjreddie.com/darknet/
21. Redmon, J., Farhadi, A.: YOLO9000: better, faster, stronger. In: IEEE Conference on Computer Vision and Pattern Recognition (2017)
22. Sahni, Y., Cao, J., Yang, L.: Data-aware task allocation for achieving low latency in collaborative edge computing. IEEE Internet Things J. **6**(2), 3512–3524 (2018)
23. Sheng, J., et al.: Computation offloading strategy in mobile edge computing. Information **10**(6), 191 (2019)
24. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:14091556 (2014)
25. Stahl, R., et al.: Fully distributed deep learning inference on resource-constrained edge devices. In: International Conference on Embedded Computer Systems, Springer (2019)
26. Szegedy, C., et al.: Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition (2015)
27. Teerapittayanon, S., McDanel, B., Kung, HT.: Distributed deep neural networks over the cloud, the edge and end devices. In: IEEE International Conference on Distributed Computing Systems (2017)
28. Tu, Y., Lin, Y.: Deep neural network compression technique towards efficient digital signal modulation recognition in edge device. IEEE Access (2019)
29. Zhao, Z., Barijough, K.M., Gerstlauer, A.: DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **37**, 2348–2359 (2018)

Authors and Affiliations

Rafael Stahl[1] · Alexander Hoffman[1] · Daniel Mueller-Gritschneder[1] · Andreas Gerstlauer[2] · Ulf Schlichtmann[1]

✉ Rafael Stahl
r.stahl@tum.de

Alexander Hoffman
alex.hoffman@tum.de

Daniel Mueller-Gritschneder
daniel.mueller@tum.de

Andreas Gerstlauer
gerstl@ece.utexas.edu

Ulf Schlichtmann
ulf.schlichtmann@tum.de

[1] Technical University of Munich, Munich, Germany

[2] University of Texas at Austin, Austin, TX, USA