Efficient Distributed Inference of Deep Neural Networks via Restructuring and Pruning

Afshin Abdi, Saeed Rashidi, Faramarz Fekri, Tushar Krishna

School of Electrical and Computer Engineering, Georgia Institute of Technology
{abdi, saeed.rashidi}@gatech.edu, {fekri,tushar}@ece.gatech.edu

Abstract

In this paper, we consider the parallel implementation of an already-trained deep model on multiple processing nodes (a.k.a. workers). Specifically, we investigate as to how a deep model should be divided into several parallel sub-models, each of which is executed efficiently by a worker. Since latency due to synchronization and data transfer among workers negatively impacts the performance of the parallel implementation, it is desirable to have minimum interdependency among parallel sub-models. To achieve this goal, we propose to rearrange the neurons in the neural network, partition them (without changing the general topology of the neural network), and modify the weights such that the interdependency among sub-models is minimized under the computations and communications constraints of the workers while minimizing its impact on the performance of the model. We propose RePurpose, a layer-wise model restructuring and pruning technique that guarantees the performance of the overall parallelized model. To efficiently apply RePurpose, we propose an approach based on ℓ_0 optimization and the Munkres assignment algorithm. We show that, compared to the existing methods, RePurpose significantly improves the efficiency of the distributed inference via parallel implementation, both in terms of communication and computational complexity.

Introduction

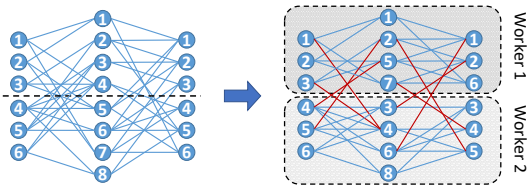
In recent years, the size and complexity of deep neural networks (DNNs) have increased significantly in terms of model's structure and number of parameters. Consequently, real-time implementation and inference in many machine learning (ML) problems has become challenging. Although the execution time of deep neural networks can be improved significantly by the application of parallel computing algorithms and using multiple processing units (such as GPU's or clusters of computing nodes), it generally requires synchronization and significant data exchange among processing units. This is mainly due to the fact that in parallel computations, each processing unit performs a portion of the computations, its inputs generally depend on the other units' outputs, and the computation results should be aggregated to yield the desired output. These co-dependencies can lead to significant delays in computations when the deep model is distributed across multiple processing nodes.

As an example, consider a sensor network where the inference is done on the data observed by the entire network, i.e., each node in the network only observes part of the data. However, transferring all data to a central powerful node to aggregate and perform the ML task is undesirable due to the sheer amount of data to be transferred over a band-limited channel, or privacy concerns. Further, such a computationally powerful node may not even exist in the network. Hence, it is favorable to develop a distributed equivalence of a deep model for distributed deployment over the sensors such that the sensor network, as a whole, becomes a computing/inference engine of the original deep model.

The majority of works on distributed/parallel execution of deep neural networks are concerned with algorithmic aspects of the parallel implementation (e.g., [41, 9, 11]). However, in the aforementioned applications, straightforward parallel computing algorithms are not suitable and cannot be arbitrarily scaled up for deep models with complex connectivity structures. Hence, here, we focus on the structure of deep models and how we can modify it for efficient parallel distributed implementation.

Although it is possible to design deep models according to the capability and constraints of the processing system, following such an approach requires training a new model for every target hardware or distributed system which is infeasible or demanding in many ML problems. Further, imposing a possibly unnecessary structure in advance during training a deep model would likely be limiting in terms of model performance and accuracy. Moreover, it can be an undesirable approach for parallel implementation since a model specifically designed for optimum implementation on a target platform or architecture may be far from optimum on other platforms (e.g., intelligent edge devices, GPUs with different compute capabilities, or CPU vs GPU vs sensor network). Hence, optimizing and fixing the structure for one particular distributed setting in advance would limit the optimal deployment on other platforms. As a result, we assume that a deep model has already been trained with minimum or no hardware-specific constraints. Our goal is readjusting the model via restructuring the layers and manipulating the parameters of the neural network without changing its general topology for more efficient parallel implementation.

For example, consider the simple neural network in Fig. 1(a). Simply partitioning the model into two sub-models



(a) Original model (b) Restructured model

Figure 1: Restructuring a neural network to reduce communication between processing units

(shown by a dashed line in Fig. 1(a)) imposes lots of communication between the two partitions. However, by rearranging the neurons properly, the co-dependency (and hence required communications) between the two sub-models (the red edges in Fig. 1(b)) is reduced substantially. It is worth mentioning that there are approximately $\mathcal{O}(P^N)$ different partitioning to distribute computations of a neural network's layer with N neurons over P workers. Hence, enumerating all such possibilities and choosing a good one is infeasible specially for large networks. In this paper, we propose a systematic approach to perform such partitioning and parameter adjustment to ensure efficient implementation of the modified model while keeping its accuracy close to the original model.

Notations- Bold lowercase letters represent vectors and the i-th element of the vector $\boldsymbol{x}$ is denoted as x_i. Matrices are denoted by bold capital letters such as $\boldsymbol{X}$, with the (i,j)-th element represented by $X_{i,j}$ or $[\boldsymbol{X}]_{i,j}$. $\boldsymbol{A} \odot \boldsymbol{B}$ is the Hadamard (element-wise) product of $\boldsymbol{A}$ and $\boldsymbol{B}$. $\|\boldsymbol{X}\|_F$ is the Frobenius norm of $\boldsymbol{X}$, $\|\boldsymbol{x}\|_2$ and $\|\boldsymbol{x}\|_0$ are the ℓ_2 and ℓ_0 norms of $\boldsymbol{x}$, respectively. $\mathbf{1}$ is a vector or matrix of all ones, whose size would be clear from the context.

Related Works

In this section, we review some of the seemingly related works and how our work differs from the existing methods.

Distributed Training- In distributed training, generally the data is split across multiple workers. There is a plethora of work on distributed deep learning with the goal of reducing communications across workers or speeding up the training (e.g, [12, 11, 8, 4, 13, 34, 28, 1]). However, a major difference with our problem is that the input to the ML model in our setting is distributed across workers, while in distributed training methods, the ML model can be fully executed on each individual worker. Moreover, the distributed training techniques are mostly focused on compressing or communicating the parameters of the models efficiently, while our primary goal is communicating the necessary information (e.g., activation signals) to speed up the inference. Similarly, our problem is different from layer-wise model partitioning for distributed training such as [20, 15] where the single-input to output latency is not a major concern and the whole input to the model is available at a single node.

Accelerating Inference on the Edge- DNN inference can impose a relatively high computation load on edge devices. On the other hand, offloading the entire inference task to the cloud may require transmitting large amount of data. To overcome these issues, recently, it has been proposed to partially run DNN at the edge and offload the remaining computation to the cloud [19, 25, 39, 36, 6]. This is generally achieved by cutting the DNN at a layer and dividing it into *two* parts, where the first part runs at the edge, and the second part is offloaded to the cloud. The output signals of the first part is then transmitted to the cloud for further processing. For example, DNN-Surgery [19] and similar works consider each layer of DNN as a whole, neither tries to break the layer into sub-layers nor restructures the model. Moreover, it is assumed that the whole data is available by the edge node.

Model Compression and Pruning- In recent years, there has been an increasing interest in compressing, quantizing, pruning, or modifying the structure of deep models to reduce their computational or storage costs, while keeping the accuracy of the modified model acceptable. The majority of these approaches can be classified into three categories:

- *Knowledge Distillation* to train a shallow or smaller model (referred to as student network) that mimics the behavior of an already trained complex model (a.k.a. teacher network) or an ensemble of teacher networks [18, 31, 38].
- *Using Structured Parameters* to reduce the size or processing time of deep model. Examples include circulant matrices [7] or adaptive Fastfood transform [37] for fully connected layers, and separable filters [30] or low-rank tensor decomposition [33] for convolutional layers.
- *Pruning Parameters* has been used extensively to reduce the complexity of the model as well as over-parametrization. ℓ_1 or ℓ_0 regularization [24], and group-sparsity [40, 35] have been successfully used to promote sparsity of the parameters during training. Model pruning algorithms such as Optimal Brain Damage [10], Optimal Brain Surgeon [16], hard-thresholding [14], and similar works [5, 23], remove the insignificant edges or neurons by considering the magnitude of the weights or their approximate Hessian matrix as a measure of importance. Further, layer-wise pruning techniques such as Net-Trim [2, 3] have been shown to be an effective tool to prune deep models while guaranteeing the accuracy of the pruned model.

Problem Statement and our Approach

Consider the problem of parallel distributed implementation of a trained deep neural network over P workers, where the deep model is divided into P sub-models, each of which is executed by a worker. As managing the synchronization and data transfer among workers degrades the efficiency of the parallel implementation, it is crucial to reduce the communication among workers. The communication is needed between the workers when the input of a neuron in a sub-model is from a neuron belonging to a different sub-model which resides in another worker. These co-dependencies can lead to significant delays in computation.

For the sake of simplicity in presentations and analysis, we mainly focus on feed-forward deep models, specifically fully-connected layers.[1] Consider an arbitrary neural network with L layers and parameters $\{\boldsymbol{\theta}^{(l)}\}_{l=1}^{L}$, where

[1] For details and the extensions of our approach to other architectures, please refer to the supplementary document.

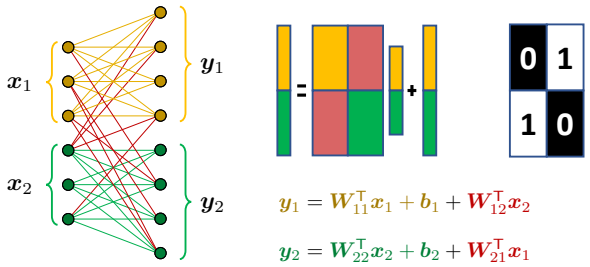


Figure 2: Parallel execution of a layer over two workers. The intra-worker computations are denoted by yellow and green, while required communication between the workers are shown by red. The binary mask matrix (right image) can be used to determine the edges between the two workers.

$\boldsymbol{\theta}^{(l)} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}$ are the weight and bias of the l-th layer. Let $\boldsymbol{x}^{(l)}$ be the input signal to the l-th layer. Then, the output of the layer (input to the next layer) would be given by

$$\boldsymbol{y}^{(l)} = (\boldsymbol{W}^{(l)})^\top \boldsymbol{x}^{(l)} + \boldsymbol{b}^{(l)}, \qquad \boldsymbol{x}^{(l+1)} = \sigma(\boldsymbol{y}^{(l)}), \quad (1)$$

where $\sigma(\cdot)$ is the activation function.

To analyze the bottlenecks, consider an arbitrary layer with input $\boldsymbol{x}$, and parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ (Fig. 2). Hence, $\boldsymbol{y} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$ would be the input signal to the neurons of the layer. Suppose that $\boldsymbol{x}_k$ and $\boldsymbol{y}_k$ are subsets of the signals that are processed by the k-th worker. Without loss of generality, we assume that the neurons are ordered such that the k-th block of consecutive neurons belongs to the k-th sub-model, i.e., $\boldsymbol{x} = [\boldsymbol{x}_1; \boldsymbol{x}_2; \dots; \boldsymbol{x}_P]$. By partitioning $\boldsymbol{W}$ and $\boldsymbol{b}$ accordingly, we observe that

$$\boldsymbol{y}_k = (\boldsymbol{W}_{k,k}^\top \boldsymbol{x}_k + \boldsymbol{b}_k) + (\sum_{l \neq k} \boldsymbol{W}_{k,l}^\top \boldsymbol{x}_l). \quad (2)$$

Note that the first term can be computed at the k-th worker independent of the others, whereas computing the second term requires synchronization and communication from the other workers. Hence, to reduce the dependency among workers and the communication cost, we consider minimizing the number of non-zero elements in $\boldsymbol{W}_{k,l}$, for $l \neq k$. Note that the bias $\boldsymbol{b}$ does not contribute to the communication between workers and can be safely ignored in computing the cost.

By defining an appropriate binary mask $\boldsymbol{M}$ (Fig. 2 (right)), the connections between sub-models is determined by the non-zero elements of $\boldsymbol{M} \odot \boldsymbol{W}$. In general, if ι_k and o_k are the number of input and output neurons assigned to the k-th worker, then $\boldsymbol{M}$ is an anti-diagonal block matrix, given by

$$\boldsymbol{M} = 1 - \text{diag}\Big(\mathbf{1}_{\iota_1 \times o_1}, \dots, \mathbf{1}_{\iota_P \times o_P}\Big).$$

Remark 1. *Note that $\|\boldsymbol{M} \odot \boldsymbol{W}\|_0$ is the total number of dependencies between sub-models, and can be used as an approximation to the total latency due to the communication and synchronization among workers. Similarly, by defining an appropriate binary mask $\boldsymbol{M}_{ij}$, the edges from worker j to i are given by the non-zero entries of $\boldsymbol{V}_{ij} := \boldsymbol{M}_{ij} \odot \boldsymbol{W}$. Depending on the communication protocol, the number of non-zero edges, number of non-zero rows, or number of non-zero columns of $\boldsymbol{V}_{ij}$ can be interpreted as a measure of latency due to the communication from worker j to i. For the sake of simplicity, in this work, we consider $\|\boldsymbol{M} \odot \boldsymbol{W}\|_0$ as the total communication latency. However, the extensions of our proposed approach to other cases is straightforward.*

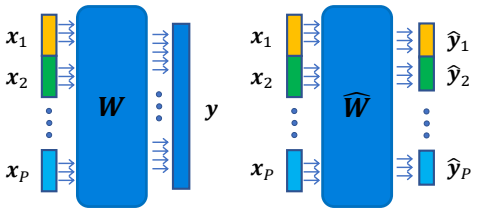


Figure 3: Rearranging neurons of a layer and adjusting parameters such that k-th worker process the k-th block, $\hat{\boldsymbol{y}}_k$.

To reduce the communication, one may attempt to naively partition the original neural network and prune the cross-edges among sub-models. However, as we observed in our experiments, there are many *important* connections between neurons from different sub-models, and naively removing these connections can severely affect the performance of the neural network. Hence, it is important to have neurons with important connections in the same sub-model. On the other hand, the problem of neuron assignment to the workers is combinatorial and discrete with complexity $\mathcal{O}(P^N)$ for a layer with N neurons and P workers. Hence, enumerating all possibilities, or using ordinary optimization techniques as well as genetic algorithms or simulated annealing would fail due to the complex nature of interactions among neurons in a deep NN. As a result, processing the entire neural network as a whole and partitioning all layers/neurons simultaneously is computationally infeasible and to the best of our knowledge, no algorithm exists to (approximately) solve the neuron assignment problem. Based on the above observations and following the success of numerous layer-wise neural network analysis algorithms, we devise *RePurpose*, a layer-wise neural network restructuring and pruning technique for efficient parallel implementation. The gist of the idea is as follows;

The neurons of the input layer are assigned to the sub-models based on each worker's computational power and/or structure of the input data. For example, in a sensor network, it is dictated by each sensor's observed data. We restructure and adjust the neural network sequentially. For the l-th layer, the assignments of the neurons in layer $l-1$ are assumed to be fixed and known from the previous steps. The neurons in layer l are rearranged and assigned to each sub-model, and the parameters of the layer are pruned and fine-tuned, such that (i) the performance of the modified neural network is close to the original one, and (ii) the communication between the sub-models (measured by the number of edges connecting neurons from different sub-models) is minimized.

RePurpose: Restructuring and Pruning Deep Models

Consider the l-th layer of neural network and assume that the neurons in the previous layers have already been partitioned

Algorithm 1: RePurpose algorithm for a single layer

Input: $\boldsymbol{W}$, $\{n_k\}_{k=1}^{P}$, η_1, η_2
Output: Permutation matrix $\boldsymbol{\Pi}$

1: Compute the cost matrix $\boldsymbol{C}$, where $[\boldsymbol{C}]_{j,i}$ is calculated via (4) and (5).
2: Construct $\widetilde{\boldsymbol{C}}$ by repeating the k-th row of $\boldsymbol{C}$, n_k times.
3: $(I, J) = \text{MUNKRES}(\widetilde{\boldsymbol{C}})$.
4: Define permutation matrix as $\boldsymbol{\Pi}_{I,J} = 1$.

and rearranged, i.e., the input of the layer is partitioned as $[\boldsymbol{x}_1; \ldots; \boldsymbol{x}_P]$, where $\boldsymbol{x}_k$ is computed at the k-th worker. Let $\boldsymbol{y}$ and $\boldsymbol{W}$ be the signals and parameters of the l-th layer in the original model. RePurpose rearranges the neurons such that the k-th block of neurons are being assigned to the k-th worker (Fig. 3). Note that the rearrangement of the neurons can be captured via a permutation matrix $\boldsymbol{\Pi}$. Hence, if we use the same weights, the effect of neuron-rearrangement can be formulated as $\widehat{\boldsymbol{y}} = \boldsymbol{\Pi}\boldsymbol{y}$ and $\widehat{\boldsymbol{W}} = \boldsymbol{W}\boldsymbol{\Pi}^{\mathsf{T}}$, and the number of cross-edges between workers would be $\|\boldsymbol{M} \odot \widehat{\boldsymbol{W}}\|_0$. To further reduce the communication between workers, RePurpose not only rearranges the neurons, but it also prunes and adjusts $\widehat{\boldsymbol{W}}$. The optimization problem is formulated as

$$\min_{\widehat{\boldsymbol{W}}, \boldsymbol{\Pi}} \|\boldsymbol{M} \odot \widehat{\boldsymbol{W}}\|_0 \qquad \text{s.t. } \|\widehat{\boldsymbol{W}} - \boldsymbol{W}\boldsymbol{\Pi}^{\mathsf{T}}\|_F^2 \leq \epsilon, \quad (3)$$

where ϵ controls the closeness of the parameters. In the following, we propose a fast and efficient method to solve (3).

Recall that if neuron i is assigned to worker j, the signal at that neuron can be rewritten as $\hat{y}_i = b_i + \widehat{\boldsymbol{w}}_i^{\mathsf{T}}\boldsymbol{x} = b_i + \widehat{\boldsymbol{w}}_{ij}^{\mathsf{T}}\boldsymbol{x}_j + \sum_{k \neq j} \widehat{\boldsymbol{w}}_{ik}^{\mathsf{T}}\boldsymbol{x}_k$, where $\widehat{\boldsymbol{w}}_i$ is the i-th column of $\widehat{\boldsymbol{W}}$, and $\widehat{\boldsymbol{w}}_{ik}$ is the k-th block of $\widehat{\boldsymbol{w}}_i$ corresponding to $\boldsymbol{x}_k$. Hence, the communication cost from other workers to worker j would be $\|\widehat{\boldsymbol{w}}_{i,\setminus j}\|_0 := \sum_{k \neq j} \|\widehat{\boldsymbol{w}}_{ik}\|_0$. By incorporating an additional optional cost to encourage the total sparsity of the parameters, $\|\widehat{\boldsymbol{w}}_i\|_0$, the cost of assigning neuron i to worker j would be

$$c_{ji} = \min_{\widehat{\boldsymbol{w}}_i} \|\boldsymbol{w}_i - \widehat{\boldsymbol{w}}_i\|_2^2 + \eta_1\|\widehat{\boldsymbol{w}}_i\|_0 + \eta_2\|\widehat{\boldsymbol{w}}_{i,\setminus j}\|_0, \quad (4)$$

where η_1 and η_2 control the trade-off between the error, sparsity, and cross-communication.

Lemma 1. *The solution of* (4) *is given by element-wise hard-thresholding of $\boldsymbol{w}_i$, i.e.,*

$$[\widehat{\boldsymbol{w}}_i]_n = \begin{cases} 0 & |\,[\boldsymbol{w}_i]_n\,| \leq \sqrt{\eta} \\ [\boldsymbol{w}_i]_n & \text{o.w.} \end{cases} \quad (5)$$

where $\eta = \eta_1$ or $\eta_1 + \eta_2$, depending on whether neuron "n" of the previous layer was assigned to the j-th worker or not.

Restructuring and neuron assignment can be interpreted as selecting elements from the cost matrix $\boldsymbol{C}$, whose (j, i)-th element is given by (4), such that (1) from row k, n_k elements are selected, i.e., n_k neurons are assigned to worker k, (2) from each column, only one element is selected, i.e., each neuron can be assigned to only one worker, and (3) the sum of selected elements is minimized, i.e., the total cost of neuron assignment and parameter adjustment is minimum.

Algorithm 1 summarizes the proposed solution, where $\text{MUNKRES}(\cdot)$ uses the Munkres assignment algorithm [21, 26] to find the (row-column) index pairs that minimizes the total sum cost $\sum_n [\widetilde{\boldsymbol{C}}]_{I_n, J_n}$.

Theorem 2. *Algorithm 1 finds the optimum solution of*

$$\|\widehat{\boldsymbol{W}} - \boldsymbol{W}\boldsymbol{\Pi}^{\mathsf{T}}\|_F^2 + \eta_1\|\widehat{\boldsymbol{W}}\|_0 + \eta_2\|\boldsymbol{M} \odot \widehat{\boldsymbol{W}}\|_0, \quad (6)$$

with computational complexity $\mathcal{O}(N^3)$, where N is the number of layer's neurons (columns of $\boldsymbol{W}$).

Note that by setting $\eta_1 = 0$, (6) would be the Lagrangian of (3) and choosing appropriate value for η_2 can lead to the desired error bound $\|\widehat{\boldsymbol{W}} - \boldsymbol{W}\boldsymbol{\Pi}^{\mathsf{T}}\|_F^2 \leq \epsilon$. Finally, it is worth mentioning that the bias term does not contribute to the communication cost and is given by $\widehat{\boldsymbol{b}} = \boldsymbol{\Pi}\boldsymbol{b}$.

Remark 2. *In model pruning and compression, it is common to fine-tune the parameters of the modified model to improve the accuracy or performance of the model. The same principle can be applied to the model obtained by the RePurpose algorithm, where the fine-tuning does not affect zeroed coefficients and hence the communication among workers.*

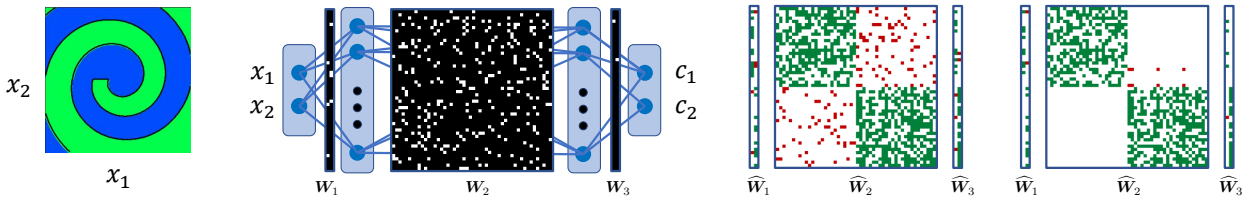
Experiments

To evaluate the performance of the RePurpose framework, we consider different DNN architectures and compare the accuracy, communication, and wall-clock times of the proposed framework to the following approaches; (1) *naive* implementation where the input data (or locally computed features) are communicated to all nodes in the network, so they all have the entire input data and process the entire deep model locally. This approach results in higher computational complexity, and possibly more communication overhead in some scenarios. (2) *baseline*: direct parallel implementation of the deep model over distributed system without any modification to the parameters or structures. Hence, there is an excessive amount of communication among workers. (3) *sparse* implementation which directly sparsifies the parameters to reduce cross-edges between the workers without rearranging the neurons. (4) *model distillation* where the input data (or features computed by the workers) are transmitted to a single worker/server for further processing. For fair comparisons, the distilled model is designed to have approximately the same computational complexity as the model obtained from RePurpose and is trained using [18]. Note that both *baseline* and *naive* methods have the same final model accuracy as they don't change the the original model.

First, We evaluate and compare the accuracy-communication trade-off in different sensor networks. Next, we investigate how the reduction in cross-communication and model simplification by RePurpose can affect the total wall-clock time in Edge networks and Data Center platforms.

Sensor Network

Setup 1. As an illustrative example, figure 4(a) shows a network of 2 sensors, sensor i observes coordinate x_i of a target object and the task is to determine whether the object is in the blue or green region. A simple fully connected neural network with 2 hidden layers of size 64 (Fig. 4(b)) is trained



(a) Classification Regions (b) Original trained model (c) $\eta_2 = 0.01$ (d) $\eta_2 = 0.1$

Figure 4: **Setup 1.** Distributed inference over a sensor network to classify location of an object. The zero coefficients in the weight matrices are represented by empty (white) spaces, inner-worker connection by green pixels and cross-worker edges by red pixels in the images. Note that for the illustration purposes, the coefficient matrix of the first layer is transposed.

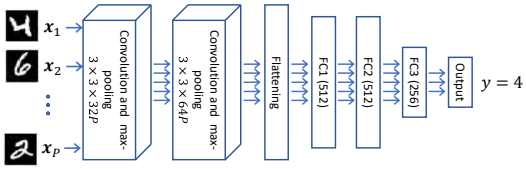


Figure 5: Structure of CNN for **Setup 2**

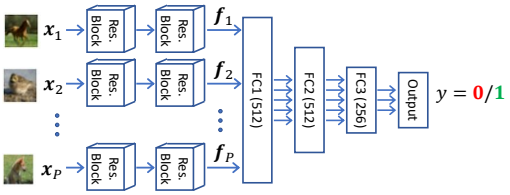


Figure 6: Structure of DNN for **Setup 3**

at a central node to perform the task with accuracy 94.5%. In the naive approach, the sensors exchange their observations (x_i's) and run the inference (NN) independently. Hence, the NN is executed twice throughout the network at the cost of higher computational complexity. Alternatively, we can apply RePurpose to efficiently distribute the inference over the sensors. We applied RePurpose with $\eta_1 = 0$, $\eta_2 \in \{0.01, 0.1\}$ (figures 4(c)-(d)). As a result, the cross-communication is reduced significantly to 1.7%, 1.5% and 1.6% for $\eta_2 = 0.01$, and 0.7%, 0.1% and 0.3% for $\eta_2 = 0.1$ for layers 1, 2, and 3, respectively. Moreover, with only 6 communicated values, the computational complexity at each sensor is reduced by almost a factor of 4 compared to the naive implementation. However, the accuracy of the RePurposed model is reduced to 93.5%. By fine-tuning the parameters, the accuracy of the RePurposed model is enhanced to 94.4%.

Setup 2. Next, we consider a network of P sensors where each sensor observes an image of a digit x_i (from MNIST dataset) and the goal is finding the rounded average $[(\sum_i x_i)/P]$. We adapted a Lenet-5 like structure [22] for the neural network which is trained in a central server (Fig. 5), and repeated the experiments several times. Note that one might attempt to classify the digits at each individual sensor and then share the values with other sensors to compute the average. However, in addition to the increased computational complexity at each individual sensor, it is worth mentioning that if the accuracy of digit recognition is ρ, close to 1, then the final accuracy in this naive approach will be reduced to approximately $\frac{1+8\rho^P}{9}$. For example, for a network with 6 sensors and $\rho = 0.98$, the final accuracy would be less than 90%. We applied RePurpose on the trained model for distributed inference over the sensor network with different communication (cross-worker edges) constraints. The results are shown in Fig. 7 for $P = 6$ sensors.

Setup 3. Next, we consider P cameras that observe different parts of a scene and detect whether a specific object exists or not. For this purpose, we used a Resnet-like neural network [17] over CIFAR10 to extract *features* of the input image locally at each node. Then, these features are communicated and processed to detect the presence of a "dog" in any of the images (Fig. 6). The simulation results are shown in Fig. 8 for a network with $P = 2$ sensors.

Figures 7a and 8a compare the performance of RePurpose with the baseline, naive sparsification, and model distillation. Clearly, RePurpose significantly outperforms sparsification. Although its accuracy is dropped for large η_2, with 1 or 10 epochs of post-training for MNIST and CIFAR10, respectively, ("FT RePurpose" compared to "FT Sparsify" in the figures) it achieves almost the same accuracy as the original model, while direct sparsification fails to provide good accuracy. On the other hand, model distillation fails to provide good accuracy, especially when the computational complexity of the model has to be small. Moreover, interestingly, RePurpose sparsifies the cross-edges between workers significantly for the hidden layers (Figures 7b and 8b), and in some situations, there is no need to transfer any data among workers for some of the hidden layers. The restructured model can achieve the same performance as the original model by using less than 0.0003 of the cross-edges (i.e., between 10 to 30 connections out of more than 100000 edges between workers). Finally, figures 7c and 8c compare the accuracy vs the cross-communication between workers. Clearly, direct sparsification performs well *only* when there are enough number of cross-edges between the workers, while the accuracy of the model obtained by RePurpose does not change for vast sparsity ranges. Finally, it is worth mentioning that in the naive approach to inference over the sensor network, each node has to transmit its observations to other nodes, hence the communication between any two pair of nodes would be 784 or 1024 values for **Setups 2** and **3**, respectively. However, RePurpose can achieve the same accuracy with less than

Table 1: Target Accelerator Evaluation Platforms

	Compute	Memory	Bandwidth
Datacenter	125 TOPS	32GB	150 GB/s (NVLink)
Edge	0.5 TOPS	1GB	100 MB/s (Ethernet)

200 total communicated values across the entire network. On the other hand, the *baseline* (direct parallelization), although achieves the same accuracy as the original model, has to transfer more than 100,000 values among workers.

System Evaluations

Methodology- We evaluate RePurpose on two distributed accelerator platforms, described in Table 1, simulated using ASTRA-sim [29]. ASTRA-sim is an open-source distributed Deep Learning platform simulator that models cycle-level communication behavior in details for any partitioning strategy across multiple interconnected accelerator nodes. ASTRA-sim takes the compute cycles for each layer of the model as an external input, and manages communication scheduling similar to communication libraries like NVIDIA NCCL [27]. We obtained compute cycles for two scenarios: (i) the Datacenter configuration from a NVIDIA V100 GPU implementation, and (ii) the Edge configuration (e.g., sensor network) from a separate DNN accelerator simulator [32].

We tried to stress the aforementioned platforms under various sized problems to show the efficiency of RePurpose and compared it with the centralized scenario where all data is gathered at a single node (server) for processing. In all models, we assumed a stack of 5 layers with same number of neurons. For the datacenter system, N varies from $1K$ to $1M$, while for edge system the variation is from $1K$ to $32K$. *We also assumed strict ordering between communication and computation, meaning that each node begins computation of each layer only when it has all inputs available*.

We picked 4 different flavors of RePurpose with 50%, 75%, 90% and 99% sparsity factor named as RP-50, RP-75, RP-90, and RP-99, respectively. In addition, we changed the number of worker nodes from 2 to 32 for both system configurations.

Remark 3. *Please note that in our evaluations, we decided to separate the hardware and model accuracy simulations, since the trade-offs are generally determined by the application, hardware, communication bandwidth, and the amount of penalty in model accuracy one might be willing to pay to speed-up the inference. However, by combining our findings in this section and the results from accuracy-communication trade-off analysis (e.g., Figures 7 and 8), one can find out the total latency of DNN inference under different scenarios and accuracies. For example, without loss of accuracy,* ***setup 2*** *and* ***3*** *can achieve 5.7 and 2.8 times speed-up over the edge network using RePurpose.*

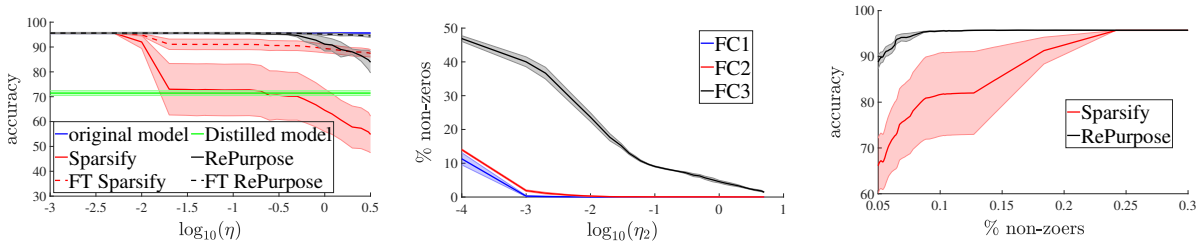
Results- Fig. 9 shows the simulation results of the communication and computation breakdown for the baseline system and RePurpose for $N = 8k$. As seen from Fig. 9a, in a datacenter system, on average and across different number of nodes, RP-50, RP-75, RP-90 and RP-99 achieve $1.7\times$, $2.76\times$, $4.77\times$ and $10.47\times$ speed-up in computations, respectively. The average improvement for communication ratio is $1.2\times$, $1.45\times$, $1.74\times$ and $1.75\times$, respectively. The reason for lower improvements of communication time is that due to NVLink's high bandwidth. For $N = 8K$, network communication time is mostly network latency limited. Hence, reduction in input size does not correspond to linear reduction in communication time.

Fig. 9b shows the similar results but for edge system. Here, due to much lower network bandwidth, the effect of communication is more considerable. On average applying RP-50, RP-75, RP-90 and RP-99 improve computation times by $1.7\times$, $2.77\times$, $4.78\times$ and $11.01\times$, respectively. This value for communication is $1.2\times$, $1.38\times$, $1.82\times$ and $3.04\times$ respectively. As the number of nodes grow, the communication gap between the baseline and RePurpose decreases. This is mostly because of the congestion in the network (e.g. switch) which signifies the importance of reducing the cross-communication among workers to speed-up the inference time.

Fig. 10 shows how communication, computation and total times change as the the number of neurons grows. For each network size, computation and communication times are averaged across different sparsity factors and node counts. For Datacenter (Fig10a), computation is the dominant factor. This is expected since the computation grows as $O(N^2)$ while communication increases as $O(N)$. In general, the total time ratio increase from $1.01\times$ at $N = 1K$ to $2.06\times$ at $N = 1M$. Communication is a more significant and considerable factor in the edge systems (Fig. 10b) due to: (i) low network bandwidth, and (ii) lower workloads on edge systems. The total time improvement for edge system is $1.55\times$ for $N = 1K$ and it increases to $3.8\times$ for $N = 32K$.

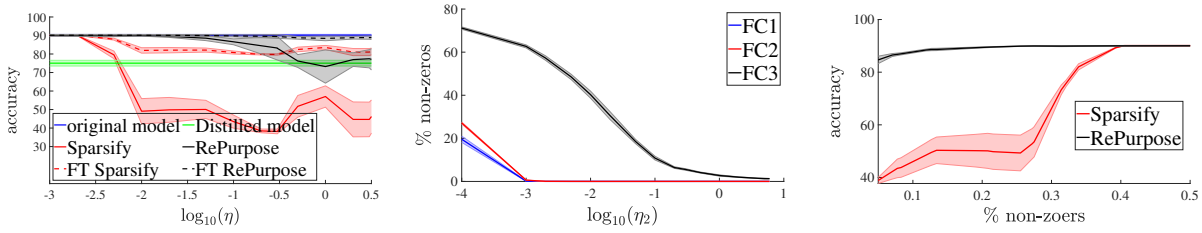
Conclusion

In this paper, we considered the problem of efficient distributed inference of an already trained deep model over a cluster of processing units or a sensor network. Required communication and synchronization among workers can adversely affect the computation time. Moreover, in the wireless sensor networks, it may significantly increase the delay and power consumption due to the transmission of large amount of data. Traditional approaches fail to consider the constraints imposed in such distributed inference systems. To overcome the shortcomings of the existing methods, we devised *RePurpose*, a framework to restructure the deep model by rearranging the neurons, optimum assignment of neurons to the workers, and pruning the parameters, simultaneously, such that the dependency among workers is reduced. Via extensive model and hardware simulations, we showed that RePurpose can significantly reduce the cross-communication between workers and improve the computation time significantly, while the performance loss of the modified model is remained negligible. Moreover, the proposed technique can reduce the computational complexity of the distributed model significantly, resulting in reduced inference time.



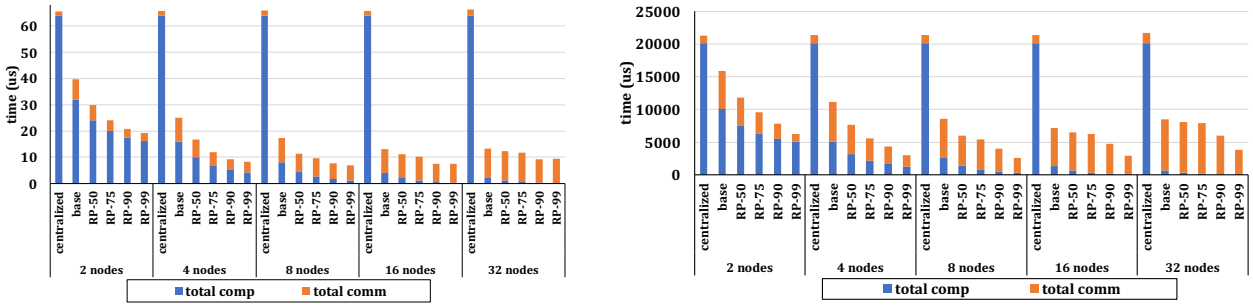
(a) Accuracy (b) % of total communications (c) Accuracy vs communication

Figure 7: RePurpose vs Sparsification and Distillation, a network with 6 nodes in **Setup 2**



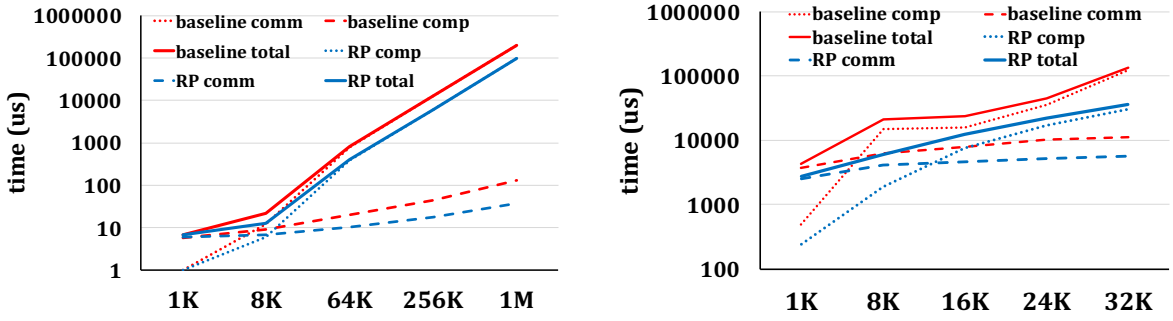
(a) Accuracy (b) % of total communications (c) Accuracy vs communication

Figure 8: RePurpose vs Sparsification and Distillation, a network with 2 nodes in **Setup 3**



(a) Datacenter Platform (b) Edge Platform

Figure 9: Communication and computation breakdown across different systems and $N = 8K$



(a) Datacenter Platform results (b) Edge Platform results

Figure 10: The effect of communication vs. computation times as the model size N grows

Acknowledgments

The research work in this paper was supported by National Science Foundation under award ID MLWiNS-2003002 and a Gift from Intel Co.

References

[1] Abdi, A.; and Fekri, F. 2020. Quantized Compressive Sampling of Stochastic Gradients for Efficient Communication in Distributed Deep Learning. In *AAAI conference on Artificial Intelligence*.

[2] Aghasi, A.; Abdi, A.; Nguyen, N.; and Romberg, J. 2017. Net-trim: Convex pruning of deep neural networks with performance guarantee. In *Advances in Neural Information Processing Systems*, 3177–3186.

[3] Aghasi, A.; Abdi, A.; and Romberg, J. 2020. Fast convex pruning of deep neural networks. *SIAM Journal on Mathematics of Data Science*, 2(1): 158–188.

[4] Alistarh, D.; Li, J.; Tomioka, R.; and Vojnovic, M. 2016. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *arXiv preprint arXiv:1610.02132*.

[5] Castellano, G.; Fanelli, A. M.; and Pelillo, M. 1997. An iterative pruning algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, 8(3): 519–531.

[6] Chen, X.; Zhang, J.; Lin, B.; Chen, Z.; Wolter, K.; and Min, G. 2021. Energy-efficient offloading for DNN-based smart IoT systems in cloud-edge environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(3): 683–697.

[7] Cheng, Y.; Felix, X. Y.; Feris, R. S.; Kumar, S.; Choudhary, A.; and Chang, S.-F. 2015. Fast neural networks with circulant projections. *arXiv preprint arXiv:1502.03436*.

[8] Chilimbi, T.; Suzue, Y.; Apacible, J.; and Kalyanaraman, K. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 571–582. ISBN 9781931971164.

[9] Chung, I.-H. H.; Sainath, T. N.; Ramabhadran, B.; Picheny, M.; Gunnels, J.; Austel, V.; Chauhari, U.; and Kingsbury, B. 2014. Parallel Deep Neural Network Training for Big Data on Blue Gene/Q. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, volume 28 of *SC '14*, 745–753. Piscataway, NJ, USA: IEEE Press. ISBN 978-1-4799-5500-8.

[10] Cun, Y. L.; Denker, J. S.; Sola, S. A.; Laboratories, T. B.; and Solla, S. A. 1990. Optimal Brain Damage. In *Advances in Neural Information Processing Systems 2*, 598–605. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1-55860-100-7.

[11] De Grazia, M. D. F.; Stoianov, I.; and Zorzi, M. 2012. Parallelization of deep networks. *Proceedings of 2012 European Symposium on Artificial NN, Computational Intelligence and Machine Learning*, 621–626.

[12] Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Mao, M.; Senior, A.; Tucker, P.; Yang, K.; Le, Q. V.; and Others. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*, 1223–1231.

[13] Dryden, N.; Jacobs, S. A.; Moon, T.; and Van Essen, B. 2016. Communication Quantization for Data-parallel Training of Deep Neural Networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, 1–8. Piscataway, NJ, USA: IEEE Press. ISBN 978-1-5090-3882-4.

[14] Han, S.; Pool, J.; Tran, J.; and Dally, W. J. 2015. Learning both Weights and Connections for Efficient Neural Networks. *CoRR*, abs/1506.02626: 1–9.

[15] Harlap, A.; Narayanan, D.; Phanishayee, A.; Seshadri, V.; Devanur, N. R.; Ganger, G. R.; and Gibbons, P. B. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR*, abs/1806.03377.

[16] Hassibi, B.; Stork, D. G.; Road, S. H.; and Park, M. 1993. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, 164–171. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1-55860-274-7.

[17] He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

[18] Hinton, G.; Vinyals, O.; and Dean, J. 2015. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*, 1–9.

[19] Hu, C.; Bao, W.; Wang, D.; and Liu, F. 2019. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. IEEE.

[20] Huang, Y.; Cheng, Y.; Chen, D.; Lee, H.; Ngiam, J.; Le, Q. V.; and Chen, Z. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv preprint, arXiv:1811.06965*, 2014.

[21] Kuhn, H. W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2): 83–97.

[22] LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324.

[23] Leung, C.-S. S.; Wong, K.-W. W.; Sum, P.-F. F.; and Chan, L.-W. W. 2001. A pruning method for the recursive least squared algorithm. *Neural Networks*, 14(2): 147–174.

[24] Louizos, C.; Welling, M.; and Kingma, D. P. 2018. Learning Sparse Neural Networks through ℓ_0 Regularization. In *ICLR*, 1–13.

[25] Mohammed, T.; Joe-Wong, C.; Babbar, R.; and Di Francesco, M. 2020. Distributed inference acceleration with adaptive DNN partitioning and offloading.

In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, 854–863. IEEE.

[26] Munkres, J. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1): 32–38.

[27] NVIDIA. 2018. NVIDIA Collective Communications Library (NCCL).

[28] Ouyang, S.; Dong, D.; Xu, Y.; and Xiao, L. 2020. Communication Optimization Strategies for Distributed Deep Learning: A Survey. *arXiv preprint arXiv:2003.03009*.

[29] Rashidi, S.; Sridharan, S.; Srinivasan, S.; and Krishna, T. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*.

[30] Rigamonti, R.; Sironi, A.; Lepetit, V.; and Fua, P. 2013. Learning Separable Filters. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE.

[31] Romero, A.; Ballas, N.; Kahou, S. E.; Chassang, A.; Gatta, C.; and Bengio, Y. 2015. Fitnets: Hints for thin deep nets. In *ICLR*.

[32] Samajdar, A.; Joseph, J. M.; Zhu, Y.; Whatmough, P.; Mattina, M.; and Krishna, T. 2020. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *IEEE International Symposium on Performance Analysis of Systems and Software*.

[33] Tai, C.; Xiao, T.; Zhang, Y.; Wang, X.; et al. 2016. Convolutional neural networks with low-rank regularization. In *ICLR*.

[34] Wang, L.; Wu, W.; Bosilca, G.; Vuduc, R.; and Xu, Z. 2017. Efficient Communications in Training Large Scale Neural Networks. In *Proceedings of the on Thematic Workshops of ACM Multimedia*, 110–116.

[35] Wen, W.; Wu, C.; Wang, Y.; Chen, Y.; and Li, H. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, Nips, 2074–2082. ISBN 1878-3686 (Electronic).

[36] Xu, Z.; Zhao, L.; Liang, W.; Rana, O. F.; Zhou, P.; Xia, Q.; Xu, W.; and Wu, G. 2020. Energy-aware inference offloading for DNN-driven applications in mobile edge clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(4): 799–814.

[37] Yang, Z.; Moczulski, M.; Denil, M.; Freitas, N. D.; Smola, A.; Song, L.; Wang, Z.; de Freitas, N.; Smola, A.; Song, L.; and Wang, Z. 2015. Deep Fried Convnets. In *The IEEE International Conference on Computer Vision (ICCV)*, 1476–1483. ISBN 9781467383912.

[38] Zagoruyko, S.; and Komodakis, N. 2017. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. In *ICLR*.

[39] Zhang, S.; Li, Y.; Liu, X.; Guo, S.; Wang, W.; Wang, J.; Ding, B.; and Wu, D. 2020. Towards real-time cooperative deep inference over the cloud and edge end devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(2): 1–24.

[40] Zhou, H.; Alvarez, J. M.; and Porikli, F. 2016. Less is more: Towards compact cnns. In *European Conference on Computer Vision*, 662–677. Springer.

[41] Zinkevich, M. A.; Smola, A. J.; Weimer, M.; Li, L.; and Smola, A. J. 2010. Parallelized Stochastic Gradient Descent. In Lafferty, J. D.; Williams, C. K. I.; Shawe-Taylor, J.; Zemel, R. S.; and Culotta, A., eds., *Advances in Neural Information Processing Systems 23*, 2595–2603. Curran Associates, Inc.