Les Cahiers du GERAD ISSN: 0711–2440

Partially-separable loss to parallellize partitioned neural network training

P. Raynaud, D. Orban, J. Bigeon

G–2023–36

August 2023

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2023
– Bibliothèque et Archives Canada, 2023

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2023
– Library and Archives Canada, 2023

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Partially-separable loss to parallellize partitioned neural network training

Paul Raynaud [a, b]

Dominique Orban [a, b]

Jean Bigeon [a, c, d]

[a] *GERAD, Montréal (Qc), Canada, H3T 1J4*

[b] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal, Montréal (Qc), Canada, H3T 1J4*

[c] *GSCOP, Université Grenoble Alpes, 38000 Grenoble, France*

[d] *Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000 Nantes, France*

paul.raynaud@polymtl.ca
dominique.orban@polymtl.ca
bigeon-j@univ-nantes.fr

August 2023
Les Cahiers du GERAD
G–2023–36

Abstract : Historically, the training of deep artificial neural networks has relied on parallel computing to achieve practical effectiveness. However, with the increasing size of neural networks, they may no longer fit into the memory of a single computational unit. To address this issue, researchers are exploring techniques to distribute the training process on powerful computational grids or less capable edge devices. In computer vision, multiclass classification neural networks commonly use loss functions depending non-linearly on all class raw scores, making it impossible to compute independently partial derivatives of weight subsets during training. In this work, we propose a novel approach for distributing neural network training computations using a *master(s)-workers* setup and a partially-separable loss function, which is a sum of *element* loss functions. Each element loss only depends on a specific subset of variables, corresponding to a subpart of the neural network, whose derivatives can be computed independently. It makes it possible to distribute every element loss and its corresponding neural network subpart across multiple workers, coordinated by one or several masters. The master(s) will then aggregate worker contributions and will perform the optimization procedure before updating the workers. To ensure that each element loss is parameterized by a small fraction of the neural network's weights, the architecture must be adapted, which is why we propose *separable layers*. Numerical results show the viability of partitioned neural networks considering a partially-separable loss function using state-of-the-art optimizers. Finally, we discuss the flexibility of a *partitioned* neural network architecture and how other deep learning techniques may reflect on it. In particular, in a federated learning context, it can preserve worker privacy, as each worker possesses only a fragment of the network, and reduce communication.

1 Introduction

The proliferation of deep neural network applications in our society is ongoing, with architecture playing a pivotal role in practical advancements. Notably, larger architectures often lead to enhanced results, as exemplified by studies such as (Cireşan, 2010). Subsequently, training computational expensive deep neural networks in reasonable time requires parallel optimization methods scaling to adequate computational resources. In the context of supervised learning, the sheer size of training datasets precludes their simultaneous evaluation. Consider computer vision, the relatively modest MNIST dataset comprises 70,000 labelled images (LeCun et al., 2010), while the vast ImageNet dataset contains over 14 million images (Russakovsky, 2015). As a result, training datasets must be subdivided into smaller minibatches, each containing a fraction of the original dataset. To parallel computations, the couples *observation-label* of a minibatch may be distributed over GPU, enabling a *data parallelism* (Cireşan, 2010; Raina et al., 2009; Dean, 2012).

Nevertheless, with the continuous expansion of neural network sizes, a solitary hardware component might prove insufficient for storing or training a neural network in isolation (Dean, 2012). This realization naturally paved the way for the concept of fragmenting neural network training, giving rise to a multitude of schemes and implementations. In the pursuit of training neural network subparts across multiple workers, two additional schemes dominate: *model parallelism* and *tensor parallelism*. In the context of model parallelism (Harlap, 2018; Huang, 2019), each worker is tasked with storing a specific neural layer. To compute the loss function and its derivatives, both forward pass and reverse passes are adapted to transfer layer computation outputs to the worker related with the subsequent layer. Conversely, tensor parallelism (Lepikhin, 2020; Bian, 2021) operates by assigning each worker a slice of one or several layers. Once all workers have computed the slices pertaining to a given layer, the results are shared among the workers to enable the next layer evaluation. Both approaches effectively fragment neural network training by incorporating communication among workers. However, in an unfavorable setup, communication costs can significantly impair practical performance (Ben-Nun and Hoefler, 2019). *Hybrid parallelism* integrates all previous strategies to propose techniques that enhance practical efficiency. For instance, both model parallelism and tensor parallelism can harness data parallelism by appropriately managing minibatches. Adapting model parallelism is straightforward, but for tensor parallelism, the same minibatch must be loaded onto multiple workers. Moreover, the workload can be shared among workers based on the computational intensity of layers. To achieve competitive performance, implementations prioritize maintaining a balanced workload across workers (Ben-Nun and Hoefler, 2019).

We introduce a novel approach to fragment deep neural network training, presenting two key contributions. The first contribution focuses on the significance of the employed loss function guiding the neural network training to compute derivatives. This holds particularly true when the loss function exhibits partial separability, manifested as a sum of non-linear element loss functions of smaller dimension. Partial separability leads to structured derivatives aggregating contributions from element loss partial derivatives computed independently. This approach lends itself to parallelization within a master(s)-workers framework by assigning one or several element functions to each worker. Unlike model parallelism or tensor parallelism, it does not necessitate communication between layers, solely requiring the aggregation of element loss contributions. Consequently, it can seamlessly integrate the parallelism schemes introduced earlier.

Effectively exploiting the advantages of partial separability during neural network training rests on the neural network's architecture. Particularly, most conventional architectures fail to exploit proficiently partial separability due to their loss element function dimensions being closely align with the overall neural network dimension. Hence, the second contribution introduces separable layers, designed to reduce loss function dimensions in comparison to the total neural network dimension. As a consequence, a partitioned neural network (i.e. stacking separable layers) can be trained over several workers containing only a subpart of the neural network. Empirical evidences combining both contributions indicates that partitioned architectures and partially separable loss functions can rival

the performance of standard architectures paired with a conventional loss function, while enabling a new distribution of computation.

The two contributions lay the foundation for a new parallelism scheme. Due to its inherent flexibility, this scheme can be applied across various contexts, spanning from grid computing to federated learning. The latter, in particular, can capitalize on the partially separable training to enhance worker privacy and reduce the need for certain communications, which remain up to now open challenges. It takes advantage of the element loss being parametrized solely by a relatively small neural network subpart to reduce the workload of edge workers, which are typically less powerful devices. Before concluding, we expose how various deep learning techniques may also be integrated within the partitioned architecture.

2 Notation table

Notation	Definition
(x, y)	a pair observation-label
(X, Y)	collection of observation-label pairs ex: a dataset, a minibatch
w	the weights parametrizing a neural network
$c_j(x; w)$	compute the raw score of the j-th class
$\mathcal{L}$	a loss function, ex: $\mathcal{L}^{\text{NLL}}$ (2), $\mathcal{L}^{\text{PSL}}$ (6a)
$h_{k,j}$	an element function of $\mathcal{L}^{\text{PSL}}$
$\widehat{\square}$	refer to $\square$ of most reduced dimension usually parametrized by $U_\square \in \mathbb{R}^{n \times n_\square}$ with $n_\square < n$ if $\square$ a function then $\square(w) = \widehat{\square}(U_\square w)$ ex: $c_j(x; w) = \widehat{c}_j(x; U_j w)$

3 Supervised learning for multiclass classification neural network

Multiclass classification neural networks have the objective of categorizing inputs into C distinct classes. To accomplish this, they calculate a score c_j for each class. The neural network determines the class involves by selecting the highest score $\arg\max_{j=1,\dots,C} c_j$. The neural network architecture is parametrized by n weights. Consequently, a neural network can be conceptualized as a function $c : \mathbb{R}^n \to \mathbb{R}^C$, applicable to any input that conforms to its architectural specifications. In the context of supervised training, the provided dataset is split into two segments: the training dataset and the test dataset. A dataset consists of a collection of observations denoted as X, paired with their corresponding labels Y. Here, each $x \in \mathbf{X}$ represents an individual observation, and its corresponding label $y \in \mathbf{Y}$ is characterized by $1 \leq y \leq C$, $y \in \mathbb{N}$. A dataset containing $\mathbf{L}$ pairs of observation-label is represented as $\{(x^{(l)}, y^{(l)})\}_{l=1}^{\mathbf{L}}$.

In conjunction with the architecture that generates the function c, the training problem necessitates an assessment of the veracity of its predictions. This role is fulfilled by the loss function $\mathcal{L}(x, y; w)$, which gauges the appropriateness of the neural network's predictions for each observation-label pair (x, y). When the loss function is applied to a minibatch $(X, Y) \subset (\mathbf{X}, \mathbf{Y})$ of size L, it means the loss of each observation-label:

$$\mathcal{L}(X, Y; w) = \frac{1}{L} \sum_{l=1}^{L} \mathcal{L}(x^{(l)}, y^{(l)}; w). \tag{1}$$

For the context of multiclass classification, the most popular loss function combines softmax layer and negative log-likelihood:

$$\mathcal{L}^{\text{NLL}}(x^{(l)}, y^{(l)}; w) = -\log \underbrace{\left(\frac{exp(c_{y^{(l)}}(x^{(l)}; w))}{\sum_{i=1}^{C} exp(c_i(x^{(l)}; w))} \right)}_{softmax}, \tag{2}$$

where $exp(c_{y^{(l)}})$ corresponds to the score of the expected result $y^{(l)}$ for the input $x^{(l)}$. The denominator of the softmax layer in $\mathcal{L}^{\text{NLL}}$ involves all the scores generated by the neural network. As a result, $\mathcal{L}^{\text{NLL}}$ is parameterized by all weights w. Consequently, calculating $\frac{\partial \mathcal{L}^{\text{NLL}}}{\partial w_i}$ during the backward pass requires knowledge of $\frac{\partial \mathcal{L}^{\text{NLL}}}{\partial w_j}$, where $j > i$ corresponds to weights closer to the output than w_i. In essence, partial derivative computations of such loss function adhere to a layer-based order. In situations where the neural network's size exceeds the capacity of a single hardware unit, model parallelism and tensor parallelism necessitates communication between workers dealing with successive layer during the backward (and forward) pass. As far as our knowledge extends, there is no technique capable of computing partial derivatives of the loss function independently and without communication for only a subset of weights distributed across all layers.

4 Partially-separable loss function

Diverging from the majority of loss functions that exhibit non-linear dependence on all scores, we investigate on partially separable loss functions. This concept, originating in the early 1980s, finds its historical roots in the partial differential equation discretized problems (Griewank and Toint, 1982; Conn et al., 1990). Subsequently, this structure has been exploited in numerous optimization areas : quasi-Newton methods (Griewank and Toint, 1982; Conn et al., 1990; Cao and Yao, 2016), derivative-free methods (Bouzarkouna et al., 2011; Price and Toint, 2006) or evolutionary methods (Durand and Alliot, 1998). A partially separable function is described as:

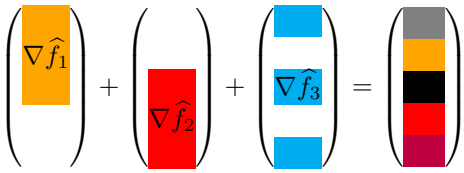
$$f(w) = \sum_{i=1}^{N} \widehat{f}_i(U_i w), \quad U_i \in \mathbb{R}^{n_i \times n}, w \in \mathbb{R}^n. \tag{3}$$

f sums element functions of smaller dimensions, denoted as $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$, where $n_i < n$. The variables that parameterize each $\widehat{f}_i$ are selects by the linear application U_i. Essentially, partial separability leads to partitioned derivatives, as illustrated by the gradient:

$$\nabla f(x) = \sum_{i=1}^{N} U_i^\top \nabla \widehat{f}_i(U_i x), \quad \nabla \widehat{f}_i \in \mathbb{R}^{n_i}, \tag{4}$$

where individual element gradients $\nabla \widehat{f}_i$ are aggregated through linear transformations $U_i^\top$ to yield the gradient ∇f. Suppose $f(w) = \widehat{f}_1(w_1, w_2, w_3) + \widehat{f}_2(w_3, w_4, w_5) + \widehat{f}_3(w_1, w_3, w_5)$ then ∇f accumulates the small contributions from $\widehat{f}_j$, $1 \leq j \leq 3$:

$$\nabla f(x) = \left(\nabla \widehat{f}_1 \right) + \left(\nabla \widehat{f}_2 \right) + \left(\nabla \widehat{f}_3 \right) = \left(\quad \right). \tag{5}$$



This characteristic generates straightforward schemes for parallelizing computations inherent to various optimization procedures (Lescrenier, 1988; Porcelli and Toint, 2022). In an adequate setup, each element function is distributed to a specific workers in a scalable manner.

It's important to note that despite the summation in (1), $\mathcal{L}^{\text{NLL}}(X, Y; w)$ is not partially separable. Specifically, $\mathcal{L}^{\text{NLL}}(x, y; w)$ does not fulfill the criteria of an element function, as its dimension matches that of $\mathcal{L}^{\text{NLL}}(X, Y; w)$, i.e., $\mathcal{L}^{\text{NLL}}(x, y; w) : \mathbb{R}^{n^{\text{NLL}}} \to \mathbb{R}$, $n = n^{\text{NLL}} \not< n$.

To define an element loss of smaller dimension, it must depend on a subset of the scores c_j, which structurally rely on a subset of weights, i.e. $\widehat{c}_j : \mathbb{R}^{n_j} \to \mathbb{R}$ parametized by U_j (Figure 1 and Figure 3). Our proposal introduces a partially separable loss (PSL), where each element loss function is formulated from a pair of scores:

$$\mathcal{L}^{\text{PSL}}(X, Y; w) := \frac{1}{L} \sum_{l=1}^{L} \sum_{j=1}^{C} e^{c_j(x^{(l)};w) - c_{y^{(l)}}(x^{(l)};w)}, \tag{6a}$$

$$= \sum_{k=1}^{C} \sum_{j=1 \neq k}^{C} h_{k,j}(X, Y; w), \tag{6b}$$

$$h_{k,j}(X, Y; w) := \frac{1}{L} \sum_{l=1}^{L} \delta_{k,j}(y^{(l)})\, e^{c_j(x^{(l)};w) - c_k(x^{(l)};w)}, \tag{6c}$$

where $\delta_{k,j}(y^{(l)}) = 1$ if $y^{(l)} = k$, and 0 otherwise. The element function $h_{k,j}$ relies solely on the weights that parametrize the two scores c_k and c_j:

$$h_{k,j}(X, Y; w) = \widehat{h}_{k,j}(X, Y; U_{k,j} w), \quad U_{k,j} \in \mathbb{R}^{n_{k,j} \times n}. \tag{7}$$

$U_{k,j}$ is the linear operator combining the weights selectioned by both U_k and U_j. As a consequence, $n_{k,j} \leq n_k + n_j$ and $n_{k,j} \leq n$. Structurally, $n_{k,j} = n_k + n_j$ when the is no weights overlap between c_j and c_k, and $n_{k,j} = n$ only if $C = 2$. If the neural network gets a maximum score c_j different from $c_{y^{(l)}}$ for a given input $x^{(l)}$, $e^{c_j(x^{(l)};w) - c_{y^{(l)}}(x^{(l)};w)}$ will return a large value.

The $\mathcal{L}^{\text{PSL}}$ loss sums $N = C^2 - C$ element functions $\widehat{h}_{k,j}$, each of which calculates a loss between every distinct class pair c_i and c_j $(i \neq j)$. In cases where the neural network is symmetric, there exists a common weights set on which every element function depends (though this set could be empty). Furthermore, all element losses perform the same computation, but with different weights as parameters—$n_{i_1,j_1} = n_{i_2,j_2}$ while $U_{i_1,j_1} \neq U_{i_2,j_2}$, for all $(i_1, j_1) \neq (i_2, j_2)$. The specific dimensions of each $\widehat{h}_{k,j}$ depends upon the layers composing the neural network architecture. The subsequent section will develop deeper this subject and introduce the concept of a *separable* layer to make $n_{k,j}$ a smaller fraction of n.

5 Separable layer and partitioned architecture

5.1 The issue of standard architectures

As previously discussed, partially separable loss functions can be employed with any type of multiclass classification neural network. For instance, Figure 1 showcases the weight dependencies of each score within an overly simplified LeNet (Lecun, 1998) architecture. This architecture incorporates two convolutional layers, consisting of two and three kernels respectively. Subsequently, two dense layers are applied. The setup is tailored for a vectorized input of size four, generating $C = 3$ class scores. Distinct colors—blue, yellow, and red—represent weights that exclusively parameterize a single score. Meanwhile, common weights shared among all scores are indicated in green. In this example, the distinct colors are solely utilized to differentiate weights pertaining to the last layer (pre-softmax (2)). All weights beneath this layer are common weights (in green). In general, when a dense layer or a convolutional layer is applied to all output neurons of a given layer, the variables preceding it in the forward pass are common for all scores c_j. As a result, any neural networks terminated by a dense layer generates scores depending on the vast majority of the network, i.e., $n_j \approx n$, $1 \leq j \leq C$. Consequently, $n_{k,j}$ tends to approach n.

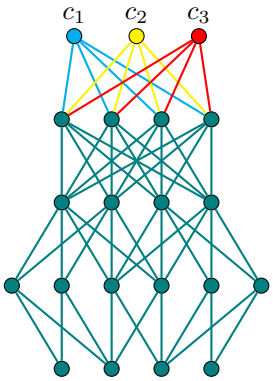


Figure 1: Weight dependency of LeNet's scores

5.2 Separable layer

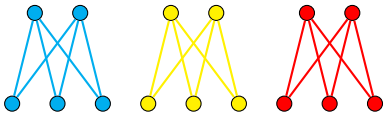


Figure 2: A separable layer, 9 x 6, considering $C = 3$ groups

To mitigate the n_j size in comparison to n, we introduce the concept of *separable* layers, depicted in Figure 2. This layer divides the neurons from two consecutive layers into C groups, with each group being fully connected to its counterpart in the subsequent layer. A separable layer employs C times fewer weights than a fully connected layer. Structurally, it propagates the group weight dependencies from one layer to its corresponding group in the next layer, to which are added the corresponding weights partition of the separable layer itself. Unlike dense or convolutional layers, it avoids distributing weight dependencies of the precedent layer across all output neurons of the subsequent layer. Relevant practical considerations include:

- If a layer's reliance non-linearly on all the outputs from a stack of separable layers, then the partitioned structure introduced by the stack vanishes.
- The output dimension of the layer preceding the initial separable layer must be a multiple of C. Consequently, dense layers will necessitate $a * C$ neurons, while convolutional layers will require $a * C$ kernels, where $a \in \mathbb{N}^*$. Each group within a separable layer in based on the output of either a neurons or a kernels.

5.3 Partitioned architecture

When multiple separable layers are successively stacked, they form a highly structured neural network which will be referred as a partitioned architecture, or more succinctly as *PSNet*. Figure 3 visualizes an architecture comprised of the same two convolutional layers as depicted in Figure 1, followed by three separable layers. Input and output configurations remain consistent Figure 1. By concentrating weight overlaps within the foundational layer, both n_j and $n_{k,j}$ become proportionally smaller in relation to n.

The $\mathcal{L}^{\text{PSL}}(X, Y; w)$ loss function achieves minimal overlap when all layers beyond the first are separable. In this ideal scenario, $U_{k_1,j_1} U_{k_2,j_2}^{\top} = 0$ holds true whenever $k_2 \neq k_1 \neq j_2$ and $k_2 \neq j_1 \neq j_2$. Moreover, each score and element loss function are parametrized respectively by $\frac{n}{C}$ and $\frac{2n}{C}$ variables, representing the smallest fractions they can attain. As the number of classes grows, the count of element loss functions $N = C^2 - C$ increases while $\frac{n}{C}$ and $\frac{2n}{C}$ decrease if the architecture favours separable layers. Additional insights of our new architecture PSNet, can be found in the numerical results section (e.g. n_j and $n_{k,j}$).

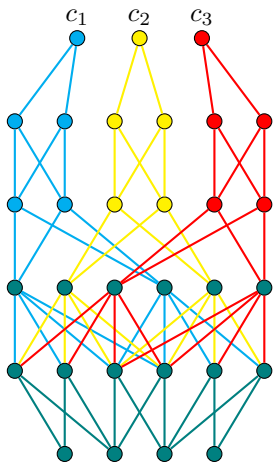


Figure 3: Weight dependency of PSNet's scores

5.4 Parallel partitioned architecture computations

In order to efficiently distribute the computation of a partitioned neural network training using a partially-separable loss function, we outline a scalable strategy. A straightforward approach is to allocate a single element function $\widehat{h}_{k,j}$ to each available worker. Since $\mathcal{L}^{\text{PSL}}$ consists of N element functions (6a), it ideally requires N workers. Each worker needs memory proportional to $\Theta(n_{k,j})$ to store its respective element function, which could be significantly smaller than n if separable layers are employed. During the forward pass, a worker computes $\widehat{h}_{k,j}$, and during the reverse pass, it computes $\nabla\widehat{h}_{k,j}$. Subsequently, the worker sends its computed contributions to the master node(s). The master node(s) maintain a vector g of size n, which is initially zeroed and accumulates worker contributions. Each worker's contribution, with size $n_{k,j}$, is scattered across g based on $U_{k,j}$. If the total network's size n is too large to be accommodated by a single hardware unit, the vector g can be managed by multiple master nodes. For instance, $C+1$ master nodes can be utilized, with one storing the common weights to all scores and the remaining C nodes storing the distinct weights of each score c_j. Additionally, the master nodes are responsible for executing the training optimizer and subsequently updating the weights of the workers.

The straightforward strategy distributing every element losses might saturate the available devices due to the quadratic growth of N with respect to C. To address this challenge, the symmetry of element losses provides a solution by allowing a worker to easily switch the element function it manages. This symmetry arises from the fact that all scores c_j share the same underlying function but are parametrized by different U_j matrices. Consequently, the results of two different element losses are governed by the same element loss function $\widehat{h}_{k,j}$, while being driven by distinct $U_{k,j}$ linear operators, i.e. $\widehat{h}_{k_1,j_1}(X, Y, U_{k_1,j_1}w) = \widehat{h}_{k_2,j_2}(X, Y, U_{k_1,j_1}w)$ where $(k_1, j_1) \neq (k_2, j_2)$. Hence, by substituting the loaded weights $U_{k_1,j_1}w$ with $U_{k_2,j_2}w$, a worker turns from $\widehat{h}_{k_1,j_1}$ into $\widehat{h}_{k_2,j_2}$ and can consecutively compute several multiple element loss functions and their derivatives. This trick significantly reduces the number of workers needed and facilitates resource utilization.

An essential point to consider is that there is no need to compute the element loss $\widehat{h}_{k,j}$ for observations $x^{(l)}$ that are not labeled as $k \neq y^{(l)}$. Therefore, the data required for a worker only represents a fraction $\frac{1}{C}$ of the entire training dataset. Additionally, several element functions may be merged, for instance, a worker can cumulate $\widehat{h}_{k_1,j_1}$ and $\widehat{h}_{k_2,j_2}$. In that case, the neural network subpart handled by the worker may expand, and the labelled observations the worker consider will combine k_1 and k_2. The only merging not augmenting the dimension is the merging of $\widehat{h}_{k,j}$ and $\widehat{h}_{j,k}$, as these two element functions weights completely overlap. Furthermore, when more than N workers are available, the same element function may be duplicated across multiple workers. In this scenario, each one of

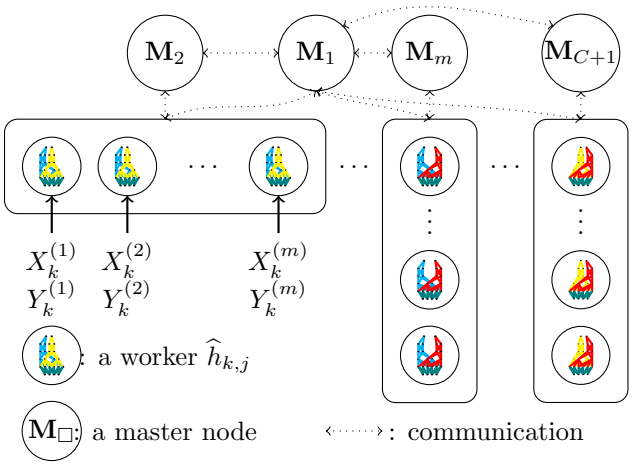


Figure 4: Partitioned neural network distribution

these workers evaluates disjoint subsets of data to ensure that duplicate computations are avoided. Finally, it is possible to combine our contributions with data parallelism, model parallelism, and tensor parallelism to further distribute computation.

Figure 4 illustrates these notions for the PSNet example from Figure 3. It denotes by $\mathbf{M}_i$ the i-th master node and by $(X_k^{(i)}, Y_k^{(i)})$ the $i-th$ minibatch from the dataset subset $(X_k, Y_k) \subset (X, Y)$ containing only the data labelled as k, i.e. $y = k,\ \forall y \in Y_k$.

6 Further improvements

6.1 Dropout consequences

The introduction of techniques like dropout (Srivastava et al., 2014) can further enhance the partial separability of the loss function. Dropout randomly deactivates some weights during the training, affecting dynamically the loss function's structure. In the context of partially-separable loss functions, this impacts the subset of weights on which each element function depends. In extreme cases where an entire layer is temporarily dropped out, the partitioned network may exhibit almost separable behavior. In Figure 5, the impact of layer dropout on a partitioned architecture is shown. Each score c_j becomes independent of all other score c_{j_2} ($U_j U_{j_2}^\top = 0$ for all $j_2 \neq j$) leading to non-overlapping element functions; therefore $U_{j_1,k_1} U_{j_2,k_2}^\top = 0$ as long as $k_2 \neq j_1 \neq j_2$ and $k_2 \neq k_1 \neq j_2$. Assuming the complete neural network can be accommodated in a single hardware, it's feasible to simultaneously compute the forward and reverse passes for multiple non-overlapping element functions. This approach requires careful selection of element functions to ensure that their weight sets do not overlap.

6.2 Federated learning application

In a federated learning context, the edge workers responsible for training the neural network possess limited computing capabilities. Furthermore, deploying the entire neural network on each worker risks potential data leakage, unless specific measures are in place. The partially-separable training approach provides a solution by allowing each worker to handle a subpart of the neural network. This not only enhances privacy but also offers the flexibility to finely adjust workloads among workers. In this setup, each element function and corresponding worker capture tendencies between pairs of classes. This leaves the worker unable to get a complete grasp on how things are entangled and apprehended in their entirety, as a complete neural network would do. Only master nodes may witness some trends,

after aggregating worker contributions. Furthermore, after each update from master(s), the neural network subparts held by workers lose their individual characteristics, enforcing worker privacy.

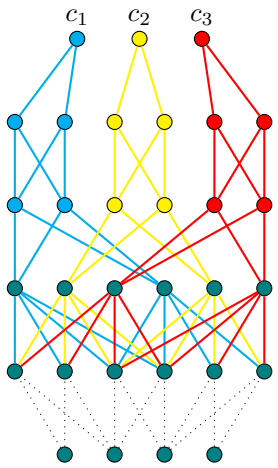


Figure 5: Weight dependency of PSNet's scores when the first convolutional layer is dropped out (dotted deactivated weights)

7 Numerical results

This section provides comparisons between LeNet-like architectures and partitioned architectures combined with either the partially-separable loss (PSL) or the negative log-likelihood loss (NLL) function. In addition the minibatch is also tested, being either 20 or 100. All pairs of architecture and loss function are trained using the same optimizer Adam (Kingma and Ba, 2017), which employs a gradient-based approach:

$$w^{(k+1)} = w^{(k)} + \alpha^{(k)} \nabla \mathcal{L}(X, Y; w), \quad \alpha^{(k)} \in \mathbb{R}^n. \tag{8}$$

Evaluating the loss function on a minibatch introduces stochastic noise to the outcomes of both the forward and reverse passes. Consequently, optimizers that train the neural network incorporate this stochastic nature by dynamically adapting $\alpha^{(k)}$. While not aiming for an exhaustive review of all stochastic gradient-based methods, we list some well-known approaches below. For more comprehensive information, interested readers are directed to surveys on this topic (Ben-Nun and Hoefler, 2019; Bottou et al., 2018). Among the most straightforward methods following equation (8) is stochastic gradient descent (Robbins and Monro, 1951; Lecun, 1998) . This was subsequently followed by the introduction of momentum by (Nesterov, 1983), and more recent methods like AdaGrad (Duchi et al., 2011), RMSProp (Hinton., 2012), and Adam (Kingma and Ba, 2017). The latter amalgamates AdaGrad and RMSProp to adjust $\alpha^{(k)}$ based on estimates of the first and second gradient moments.

The subsequent results serve as a proof of concept to demonstrate the effectiveness of the partitioned network and partially-separable loss. During the submission period, our resources and expertise did not extend to testing this partitioned structure within an extensive distributed framework. Consequently, the training are run on Nvidia A100 Tensor Core GPU and involve relatively small architectures. Our near future plans entail the implementation of a distributed training version once we establish collaboration with an interested industrial partner.

Our focus centers on two specific datasets: MNIST (LeCun et al., 2010) and CIFAR10 (Krizhevsky, 2009), both encompassing ten distinct classes ($C = 10$). MNIST comprises grayscale images of handwritten digits, each having dimensions of 28×28 pixels. Conversely, CIFAR10 consists of color images (i.e. three channel inputs) with dimensions of 32×32 pixels. For generating numerical results, the code is implemented using Julia programming language (Bezanson, 2017). The architectures presented in both figures are constructed using Knet.jl (Yuret, 2020), while the datasets are sourced

from MLDatasets.jl (L. and S., 2016). The code producing these numerical results is publicly available at https://github.com/JuliaSmoothOptimizers/KnetNLPModels.jl/ on the branch pr-AAAI24-scripts. Both Figure 6 and Figure 7 compile the trainings of architecture-loss pairs: PSNET-NLL, PSNET-PS, LeNet-NLL and LeNet-PS. The curves display the mean of 10 trainings and are surrounded by their respective standard deviation errors.

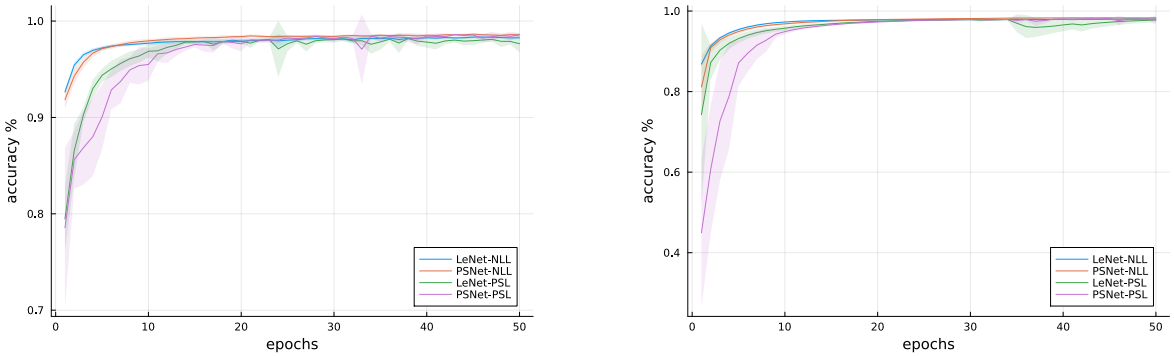


Figure 6: LeNet and PSNet training accuracies over epochs on MNIST. The left (resp. right) figure considers minibatch of size 20 (resp. 100).

In Figure 6, both architectures are tailored for the MNIST dataset. The LeNet architecture comprises two convolutional layers followed by three dense layers. Both convolutional layers employ 5×5 kernels and incorporate average pooling. The first convolutional layer employs 6 kernels, while the second employs 16 kernels. The subsequent fully connected layers contain 120, 84, and 10 neurons respectively.

On the other hand, the partitioned network, denoted as PSNet, also consists of two convolutional layers followed by three separable layers. Both convolutional layers use 5×5 kernels and include average pooling. The first convolutional layer employs 40 kernels, while the second employs 30 kernels. As a result, the separable layers consist of 240, 150, and 10 neurons respectively. The LeNet architecture is parameterized by 44426 weights, while the PSNet architecture is parameterized by 53780 weights. In PSNet, each score (resp. element loss) relies on 6340 (resp. 11588) weights. All scores share a common foundation of 1092 weights at the root of the neural network.

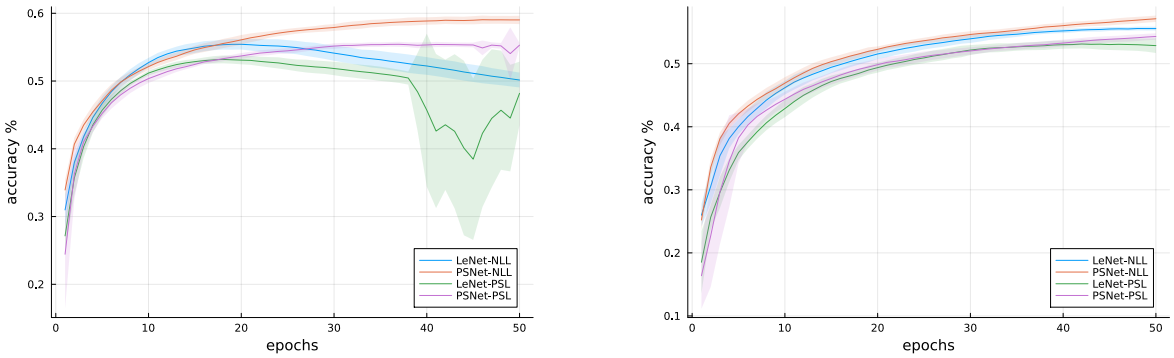


Figure 7: LeNet and PSNet training accuracies over epochs on CIFAR10. The left (resp. right) figure considers minibatch of size 20 (resp. 100).

In Figure 7, the architectures depicted in the first figure have been modified to accommodate input sizes suitable for the CIFAR10 dataset. Both LeNet and PSNet have adjusted their first convolutional layers to handle three-channelled images. Additionally, the three dense layers of LeNet have been adapted to consist of 200, 100, and 10 neurons, respectively. In the case of PSNet, the number of kernels in the first convolutional layer has been increased from 40 to 60. Furthermore, the separable

layers have been adjusted to be parameterized by 350, 150, and 10 neurons. With these adaptations, the modified LeNet and PSNet architectures are parameterized by 103882 and 81750 weights, respectively. In PSNet, each score (resp. element loss) depends on 12279 (resp. 19998) weights. In that case, the element loss function share 4560 common weights.

The comparison results presented in Figure 6 showcase the accuracy of the loss function PSL or NLL to train LeNet and PSNet onto the MNIST dataset. Regardless of the architecture employed, any training using the partially-separable loss (PSL) starts off slower compared to the negative log-likelihood (NLL) loss, especially when using small minibatch sizes like 20. However, over time, PSL training tends to catch up with NLL training and approaches asymptotically similar performances. The standard deviation of training results is higher for PSL compared to NLL, a pattern that is consistent across both datasets, as demonstrated in Figure 7. Figure 7 displays the same comparisons for the CIFAR10 dataset. While the ranking between loss functions remains consistent, architecture changes play a more significant role than the choice of loss function. This phenomenon is noticeable for minibatch sizes of 100 and is particularly evident for minibatch sizes of 20. When considering minibatches size of 20, the training of LeNet-PSL exhibits high standard deviation errors. Raw results indicate consistent drops in accuracy (down to 20%), the accuracy tends to be regained after a few epochs, recovering a stable 50%. Investigating this behavior is a topic for future research.

Given that PSNet is composed of fewer weights than LeNet (81750 vs. 103882), it is surprising to witness PSNet architecture dominate the adapted LeNet. This difference can be attributed to the use of separable layers in PSNet instead of dense layers, significantly reducing the number of weights required to accommodate PSNet to the CIFAR10 dataset. Hence, to compare fairly LeNet with a PSNet architectures having similar amount of weights, we added kernels on the convolutional layers building PSNet to compensate the dimension growth of LeNet. Kernels are cornerstones of the computer vision current success. As a consequence, we believe their multiplication in the adapted PSNet are likely to overcome LeNet performances.

8 Conclusion

We have introduced a novel parallelization approach exploiting partially-separable loss functions applied to partitioned neural networks. This method offers flexibility and scalability, complementing existing deep learning distributed computing techniques. In distributed learning scenarios, it reduces the dimensionality of worker's computation without increasing communication. The structure of each worker's neural network subpart helps to preserve the privacy of edge device data. Moreover, the promising numerical results could be further improved by incorporating adaptations of other deep learning techniques such as dropout or quasi-Newton training. In future research, we plan to implement partially-separable training within distributed frameworks and explore asynchronous training, which lead to structured communication patterns among workers. Additionally, we intend to investigate the unexpected behavior of PSNet-NLL and its potential connections to gradient propagation.

References

T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Comput. Surv., 52(4), aug 2019. doi: 10.1145/3320060. URL https://doi.org/10.1145/3320060.

J. et al. Bezanson. Julia: A fresh approach to numerical computing. SIAM Review, 59(1):65–98, 2017. doi: 10.1137/141000671.

Z. et al. Bian. Maximizing parallelism in distributed training for huge neural networks, 2021.

L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. SIAM Review, 60(2):223–311, 2018. doi: 10.1137/16M1080173. URL https://doi.org/10.1137/16M1080173.

Z. Bouzarkouna, A. Auger, and D. Ding. Local-meta-model CMA-ES for partially separable functions. In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO '11, pages 869–876, New York, NY, USA, 2011. doi: 10.1145/2001576.2001695.

H. Cao and L. Yao. A partitioned PSB method for partially separable unconstrained optimization problems. 290:164–177, 2016. doi: 10.1016/j.amc.2016.06.009.

D. C. et al. Cireşan. Deep, big, simple neural nets for handwritten digit recognition. Neural Computation, 22 (12):3207–3220, dec 2010. doi: 10.1162/neco_a_00052. URL https://doi.org/10.1162%2Fneco_a_00052.

A. R. Conn, N. I. M. Gould, and Ph. L. Toint. An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. Computing Methods in Applied Sciences and Engineering, pages 42–54, 1990. doi: 10.1007/BF02592099.

J. et al. Dean. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, Advances in Neural Information Processing Systems, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. 12, 2011. doi: 10.5555/1953048.2021068.

N. Durand and J-M Alliot. Genetic crossover operator for partially separable functions. In GP 1998, 3rd annual conference on Genetic Programming, Madison, United States, 1998. URL https://hal-enac.archives-ouvertes.fr/hal-00937718.

A Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. 39:119–137, 1982. doi: 10.1007/BF01399316.

A. et al. Harlap. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

G. Hinton. Neural networks for machine learning, lecture 6a: Overview of mini-batch gradient descent. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, June 2012.

Y. et al. Huang. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

Alex Krizhevsky. Learning multiple layers of features from tiny images. pages 32–33, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Carlo L. and Christof S. Mldatasets.jl:, 2016. URL https://github.com/JuliaML/MLDatasets.jl.

Y. LeCun, C. Cortes, and C.J. Burges. Mnist handwritten digit database. ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist, 2, 2010.

Y. et al. Lecun. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11): 2278–2324, Nov 1998. doi: 10.1109/5.726791.

D. et al. Lepikhin. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

M. Lescrenier. Partially separable optimization and parallel computing. 14(1):213–224, 1988. doi: 10.1007/BF02186481.

Y. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. Proceedings of the USSR Academy of Sciences, 269:543–547, 1983. URL https://www.mathnet.ru/eng/dan46009.

M. Porcelli and Ph. L. Toint. Exploiting problem structure in derivative free optimization. ACM Trans. Math. Softw., 48(1), feb 2022. doi: 10.1145/3474054.

C. J. Price and Ph. L. Toint. Exploiting problem structure in pattern search methods for unconstrained optimization. 21(3):479–491, 2006. doi: 10.1080/10556780500137116.

R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, page 873–880, New York, NY, USA, 2009. Association for Computing Machinery. doi: 10.1145/1553374.1553486. URL https://doi.org/10.1145/1553374.1553486.

H. Robbins and S. Monro. A Stochastic Approximation Method. The Annals of Mathematical Statistics, 22 (3):400–407, 1951. doi: 10.1214/aoms/1177729586. URL https://doi.org/10.1214/aoms/1177729586.

O. et al. Russakovsky. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

D. et al. Yuret. Knet.jl: Koç university deep learning framework., 2020. URL https://github.com/denizyuret/Knet.jl.