Future Generation Computer Systems 149 (2023) 432–444

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Layer-wise partitioning and merging for efficient and scalable deep learning

S.B. Akintoye[a], L. Han[a,*], H. Lloyd[a], X. Zhang[a], D. Dancey[a], H. Chen[b], D. Zhang[c]

[a] Department of Computing and Mathematics, Manchester Metropolitan University, UK
[b] Department of Computer Science, University of Sheffield, UK
[c] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, PR China

ARTICLE INFO

Article history:
Received 28 July 2022
Received in revised form 19 July 2023
Accepted 30 July 2023
Available online 3 August 2023

Keywords:
Deep Neural Network
Deep learning
Data parallelism
Model parallelism
Parallel and distributed computing
High performance computing

ABSTRACT

Deep Neural Network (DNN) models are usually trained sequentially from one layer to another, which causes forward, backward and update locking problems, leading to poor performance in terms of training time. The existing parallel strategies to mitigate these problems provide suboptimal runtime performance. In this work, we have proposed a novel layer-wise partitioning and merging, forward and backward pass parallel framework to provide better training performance. The novelty of the proposed work consists of (1) a layer-wise partition and merging model which can minimise communication overhead between devices without the memory cost of existing strategies during the training process; (2) a forward pass and backward pass parallelisation to address the update locking problem and minimise the total training cost. The experimental evaluation on real use cases shows that the proposed method outperforms the state-of-the-art approaches in terms of training speed; and achieves almost linear speedup without compromising the accuracy performance of the non-parallel approach.

1. Introduction

Deep Neural Networks (DNNs) have shown promise in different applications such as computer vision, Natural language processing and speech recognition. However, training a DNN remains a significant challenge, which is both computational and data intensive [1–3]. To mitigate this problem, DNN models are usually trained in parallel across either homogeneous or heterogeneous devices including CPUs and GPUs [4] for better training performance. One of the common distributed training methods is model parallelism [5]; model parallelism allocates disjoint subsets of a DNN model to each dedicated device [6]. This method requires data communication between computation processes to update the model in each training iteration. The backpropagation algorithm [7,8] is usually used for the updates and consists of two phases: the forward pass and backward pass. The forward pass calculates and stores intermediate variables such as outputs for a neural network from the input to the output layer. The backward pass method calculates the gradients of neural network parameters, in reverse order, from the output to the input layer. The sequential execution of forward pass and backward pass requires data communication between computation processes to update the model in each training iteration, usually referred to as forward, backward, and update locking problems [9], which lead to inefficient training performance due to computation dependencies.

Several methods have been proposed to mitigate these problems. One of these methods is *delayed gradients*, which breaks the backward locking [10,11]. However, this method suffers from large memory usage due to the requirement to store all the intermediate computation results. In addition, the delayed gradients method provides suboptimal performance in terms of training speed and convergence rate when the DNN model becomes deeper and larger. Another method is *feature replay* [12,13], which also breaks backward locking and provides better performance than delayed gradients in memory consumption. The main disadvantage is that feature replay has a greater computational load than delayed gradients, leading to lower training speed. Finally, *layer-wise parallelisation* is a method in which each network layer is parallelised individually, with the solution to a graph search problem used to optimise the layer parallelisation [14]. However, this method still incurs communication overhead because the computations of each layer are performed on a single device while the entire model is trained on multiple devices using the data parallelism technique, which requires sharing gradients across devices consequently limiting training performance.

Unlike existing methods, this paper proposes a novel layer-wise partitioning and merging approach to minimise communication overhead between devices without incurring a significant memory overhead during the training process. Different from the existing approaches, our proposed method applies both partition

[*] Corresponding author.
E-mail address: l.han@mmu.ac.uk (L. Han).

and merging operations to perform computations of network layers across available multiple devices to minimise the communication overhead. In addition, we propose a forward pass and backward pass parallelisation method to address the update locking problem associated with the sequential execution of forward pass and backward pass computations. Thus, the main contributions of this paper include:

- We propose a novel layer-wise partitioning and merging for efficient distribution and processing of network layer computations across multiple devices. The partitioning and merging mechanism can minimise communication overhead between devices in distributed training.
- We propose a forward pass and backward pass parallelisation method for solving locking problems, with an associated cost function formulation for optimising training performance by reducing the total training cost.
- We apply the proposed methods to two real use cases representing different complexity of the models for performance evaluation of the proposed approach.

The remaining parts of this paper are organised as follows. In Section 2, we summarise the related research work to this paper. In Section 3, we introduce a layer-wise partitioning and merging, forward pass and backward pass parallelisation framework. We conduct experiments to evaluate our proposed method in Section 4, and Section 5 concludes the work and highlights the future work.

2. Related work

The increase in dataset and DNN model sizes has motivated the use of distributed training of DNN models for better performance. Existing parallel methods are usually developed based on the *data* and *model parallelism* techniques to distribute training across multiple devices. Data parallelism divides the entire training dataset into subsets of data and dispatches on multiple devices. Each device maintains a DNN model replica and its parameters. On the other hand, model parallelism splits and trains large DNN models onto multiple computation devices instead of a single device for efficient training performance [15,16]. In [17], data parallelism was used for convolutional and pooling layers and model parallelism for densely connected layers to accelerate CNNs training performance. Wu et al. [18] adopted data parallelism to allocate the RNN model replica on each node and model parallelism for intra-node parallelisation. Although these works improve performance over either data or model parallelism, they still provide suboptimal performance and scale poorly on large datasets and multiple devices. Saguil and Akramul [19] proposed a layer partitioning method to improve the training performance of neural network-based embedded applications in edge networks. The method partitions layers of a model into sub-models and distributes them among different devices. The method was shown to reduce the communication overhead between devices by up to 97% with a tradeoff of 3% in accuracy. Similarly, the works in [20–24] split neural networks layers and allotted sublayers to multiple devices for improved training performance. Finally, Song et al. [25] proposed layer-wise parallelism which partitions feature map tensors, kernel tensors, gradient tensors, and error tensors, subsequently optimising the partition with the goal of minimising the total communication for the acceleration of the DNN training.

Aside from the data and model parallelism, *pipeline parallelism* [26–28], and *hybrid parallelism* [25,29–31] have been proposed to speed up DNN training further. Pipeline parallelism partitions model layers into stages and runs them on multiple devices. Huang et al. [32] proposed *GPipe*, a pipeline parallelism based solution that explores the synchronous approach to train large models and optimised GPU memory usage. Narayanan et al. [33] proposed *PipeDream*, which uses the hybrid method of data and pipeline parallelism for asynchronous training of the DNN models. Hybrid parallelism combines the advantages of two or more types of parallelism while weakening the disadvantages of each for better performance.

In recent times, more parallel strategies have been proposed to improve the training performance by addressing forward, backward, and update locking problems [9]. Belilovsky et al. [34] proposed a greedy algorithm based solution known as *Decoupled Greedy Learning* (DGL) to achieve update unlocking as well as forward unlocking. The work decoupled and parallelised the CNN layers training to achieve better convergence performance than state-of-the-art approaches. Furthermore, Huo et al. [35] proposed a *Decoupled Parallel Back-propagation* (DDG, in which the DG refers to delayed gradients), which splits the network into partitions and solves the problem of backward locking by storing delayed error gradient and intermediate activations at each partition. Similarly, Zhuang et al. [11] adopted the delayed gradients method to propose a fully decoupled training scheme (FDG). The work breaks a neural network into several modules and trains them concurrently and asynchronously on multiple devices. In DDG and FDG, the forward pass executes sequentially. The input data flows from one device to the other and computes sequential activation order. On the other hand, all devices except the last one store delayed error gradients and execute the backward computation after the forward computation is complete. DDG and FDG adopt the delayed gradients to split the backward pass and reduce the total computation time to $T_f + T_b/N$, where T_f, T_b, and N denote forward pass time, backward pass time, and the number of devices for a mini-batch in Naive sequential method. However, the two delayed gradients based approaches incur a large memory overhead due to storing intermediate results and suffer from weight staleness and the forward locking problem [9].

To address these challenges, Xu et al. [36] proposed *Layer-wise Staleness* and *Diversely Stale Parameters* (DSP), a combination of parallel DNN model training algorithms, where ‘staler' information is used to update lower layer parameters. DSP overlaps both forward pass and backward pass computations to the reduce memory consumption experienced in DDG and FDG during the training process, improving training performance. However, the staleness-based methods have slow convergence, especially when the DNN models become more complex and deeper, thereby negatively impacting training performance in terms of speed for the desired accuracy [37].

To address the limitations of the aforementioned parallel methods, we propose a novel layer-wise partitioning and merging, forward pass and backward pass parallelisation approach for accelerating distributed training of the DNN models.

3. The proposed method

In this section, we provide full details of the proposed method, which divides into two phases: layer-wise partitioning and merging; and forward pass and backward pass parallelisation. The proposed method aims to minimise the communication overhead and address locking problems associated with the sequential execution of forward pass and backward pass computations in a distributed environment.

3.1. Problem statement

In deep learning, backpropagation is an algorithm to train feedforward neural networks, divided into forward pass and backward pass phases for sequential computations of activation

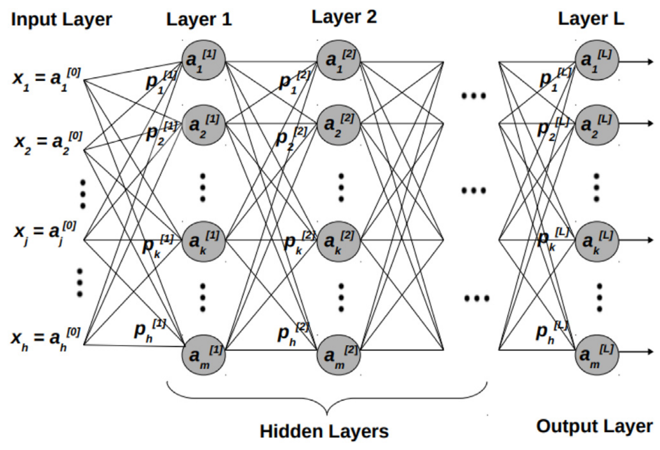


Fig. 1. Feedforward neural network.

Table 1
Notation used in this paper.

Notations	Descriptions
L	Number of network layers.
p^l	The weight parameter of l layer.
a^l	The activation of layer l.
$X = a^{[0]}$	The input data.
e	The input-label.
b	The Batch size.
f	The Loss function.
m	Number of layer computations.
Z	Sub-modules from similar or different layers.
$z^{[j]}$	Sub-module such that $z \in Z$ and $j = \{1, 2, \ldots, n\}$
q_t	Parameter of u and v at iteration t.
n	Number of GPUs.
G	The activation function.
ω^t	Gradient of the function at iteration t.
T_F	Forward pass time.
T_B	Backward pass time.

and error gradient, respectively. However, the sequential calculations result in backward locking and forward locking problems due to computation dependencies between network layers. In addition, there is also an update locking problem because the backward pass waits for the forward pass to finish before it starts.

The process of training feedforward neural networks is represented in Fig. 1, and Table 1 presents the notation used here for the training parameters.

We assume that a DNN model has L consecutive layers and $p = (p^{[1]}, p^{[2]}, \ldots, p^{[l]}) \in \mathbb{R}^h$ where $p^{[l]} \in \mathbb{R}_l^h$ denotes the weight parameter at layer $l \in \{0, 1, \ldots, L-1\}$ and $h = \sum_{l=0}^{L-1} h_l$. The activation of each layer l is defined as:

$$a^{[l+1]} = g_l(a^{[l]}, p^{[l]}) \tag{1}$$

where $a^{[l]}$ and $a^{[0]}$ are the input of layer l and input data respectively. Generally, the layer's activation value can be defined as:

$$\begin{aligned} a^{[l+1]} &:= G(a^{[1]}, p^{[1]}, p^{[2]}, \ldots, p^{[l]}) = \\ &g_l(\ldots g_2(g_1(a^{[1]}, p^{[1]}), p^{[2]} \ldots, p^{[l]})) \end{aligned} \tag{2}$$

The loss function is $f(a^{[L]}, e)$, where e denotes input-label detail of the training samples. The loss function of the feedforward neural network can be represented as the following optimisation problem:

$$\min_f g(p) = f(G(a^{[1]}, p^{[1]}, p^{[2]}, \ldots, p^{[l]}), e) \tag{3}$$

Gradient descent is used to solve the optimisation problem given in Eq. (3) by iteratively moving in the direction of the negative of the gradient of the function at iteration t is defined as:

$$\omega_p^t = [\omega_{p^{[1]}}^t, \omega_{p^{[2]}}^t, \ldots, \omega_{p^{[l]}}^t] \tag{4}$$

where,

$$\omega_{p^{[l]}}^t = \frac{\delta g(p_t)}{\delta p_t^{[l]}} \tag{5}$$

Typically, either stochastic gradient descent (SGD) [38] or more recent algorithms such as ADAM [39] are used to update the model parameters p iteratively as:

$$p_{t+1}^{[l]} = p_t^{[l]} - \alpha_t \omega_{p^{[l]}}^t \tag{6}$$

where α_t is the learning rate. When the training sample is large and b_t is mini-batch of b, the gradient vector becomes:

$$\omega_{p^{[l]}}^t = \frac{\delta g_{b_t}(p_t)}{\delta p_t^{[l]}} \tag{7}$$

The backpropagation algorithm is usually used to calculate the model gradients, which consists of two processes: the forward pass for model prediction and the backward pass for gradient calculation and model update. In the backpropagation process, the input of each layer relies on the output of the immediate previous layer. For instance, the gradient in layer l using the gradient back-propagated from layer u and v such that $l < u < v$ can be expressed as:

$$\omega_{p^{[l]}}^t = \frac{\delta g_{b_t}(p_t)}{\delta p_t^{[l]}} = \frac{\delta q_t^{[u]}}{\delta p_t^{[l]}} \frac{\delta g_{b_t}(p_t)}{\delta q_t^{[u]}} = \frac{\delta q_t^{[u]}}{\delta p_t^{[l]}} \omega_{q^{[u]}}^t \tag{8}$$

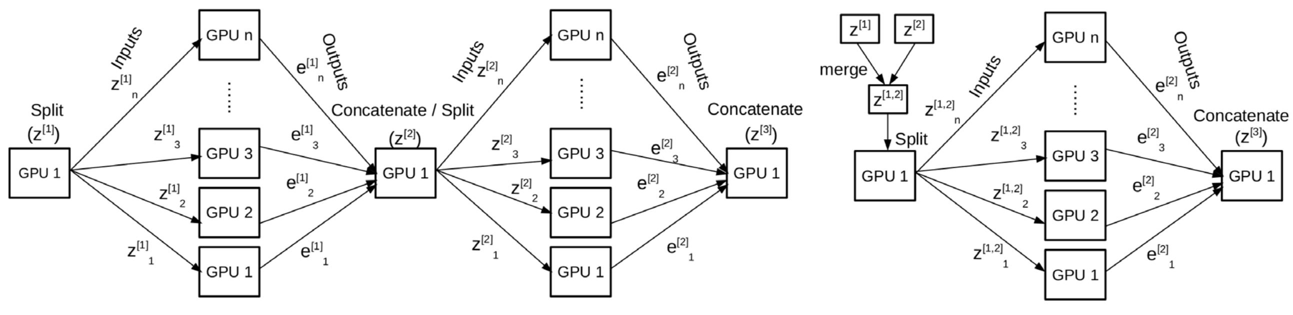
where,

$$\omega_{q^{[u]}}^t = \frac{\delta g_{b_t}(p_t)}{\delta q_t^{[u]}} = \frac{\delta q_t^{[v]}}{\delta q_t^{[u]}} \frac{\delta g_{b_t}(p_t)}{\delta q_t^{[v]}} = \frac{\delta q_t^{[v]}}{\delta q_t^{[u]}} \omega_{q^{[v]}}^t \tag{9}$$

Moreover, the backward process waits until the forward process is complete. This situation, often referred to as the forward, backward and update lockings problem, causes delays in the model updates, leading to poor training performance in terms of training speed.

3.2. Layer-wise partitioning and merging

To mitigate the aforementioned problem, we propose a novel layer-wise partitioning and merging method. The method uses layer-wise partitioning and merging operations to solve forward and backward locking problems by performing computations of network layers across available multiple devices rather than a single device as in the existing layer-wise partitioning methods. The layer-wise partitioning has two levels. In the first level, given a DNN consisting of a number of layers (i.e. dimension), we break the computations of those layers with high computational load, such as convolution layers, based on a number of available devices D (such as GPUs). In the case of convolution layers, the kernel filter slides over the input matrix to get the output vector. For instance, if the input matrix has dimensions of M_i and M_j, and the kernel matrix has dimensions of E_i and E_j, then the final output will have a dimension of $M_i - E_i + 1$ and $M_j - E_j + 1$. The computations to get values of the final output are divided equally among the available number of GPU n. However, in rare cases where the number of computations m in the dimension is less than the available number of GPUs n, i.e. $m < n$, the computations will be allocated to m GPUs, and the remaining GPUs $n - m > 0$ will be left unused. We therefore define $n_l = \min(m, n)$, the number of GPUs over which the layer l is parallelised. The second level regroups the sub-layers and distributes them equally into sub-modules based on the number of available GPUs n_l to ensure



(a) Layer partitioning and merging

(b) Sub-modules merging and splitting

Fig. 2. Layer partitioning and merging using n number of GPUs.

a well-balanced workload across multiple devices to achieve the best possible runtime performance while maintaining the original network accuracy. As shown in Algorithm 1, we split the convolution layer $p^{[1]}$ into $\{p_1^{[1]}, p_2^{[1]}, \dots, p_{n_l}^{[1]}\}$, where $n_l = \min(m, n)$ and n is the cardinality of the set of available devices D, where $D = d_i, \quad i \in [1, \dots, n]$. Subsequently, we partition the new set of DNN layers into Z sub-modules across the number of available devices n in which z each sub-module comprises a stack either from similar or different layers. For instance, if the number of available devices is three, i.e. $n_l = 3$, and the number of the new set of DNN layers is seven, which includes sub-layers of a partitioned layer and non-partitioned layers, the algorithm groups seven layers equally into the three sequentially, i.e., $Z = \{z_1, z_2, z_3\}$, number of available devices. In this case, the last sub-module gets the remainder of the layers division, i.e., z_1 gets two layers, z_2 gets two layers and z_3 gets three layers.

In Fig. 2(a), each GPU_i, also represented as d_i computes each partition z_i^j of sub-module $z^j \in Z$, where $i \leq n$ and $j = \{1, 2, \dots, n\}$. The outputs are concatenated and re-partitioned for the next sub-module $j+1$. GPU 1, which serves as a proxy server, does the merging operation, splitting computations across the available GPUs, and concatenates activation outputs from the GPUs for the next splitting operation. In addition, the sub-module z performs a forward and a backward pass using activation input and gradient from module $z-1$, respectively. From Eq. (1), in the forward pass for at iteration t, the activation a_{z-1}^t of $z-1$ is used as input to the sub-module z and produces activation a_z^t. Likewise, from Eq. (6), in the backward pass at iteration t, the gradient from sub-module $z-1$ is fed input into sub-module z to produce a new gradient.

However, this method incurs high communication overhead due to frequent data movement among the devices. To address this challenge, we merge two or more sub-modules as shown in Fig. 2(b) such that the output of a sub-module is sent directly to the next sub-module without involving device computation; this method also ensures even distribution and all layers benefit from n GPUs.

Algorithm 1: Layer partitioning and merging

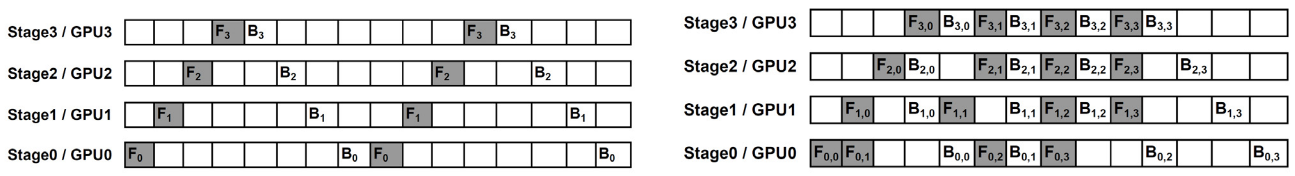
```
input : $p^{[l]}$: weight parameter of $l$ layer
        $L$: Number of network layers.
        $n$ : no. of available GPUs
        $D = d_i, \quad i \in [1, \dots, n]$: list of GPUs
output: $Z$: list of sub-modules
1  set $Z$ to {} ;
2  for $l \leftarrow 1$ to $L$ by 1 do
3      if dimension of layer $l > 1$ then
4          $m \leftarrow$ no. of computations;
5          $n_l \leftarrow \min(m, n)$;
6      end
7  end
8  $K \leftarrow$ number of layers (partitioned and non-partitioned),
     $K \geq L$ ;
9  group the $\{p_1^{[l]}, p_2^{[l]}, \dots, p_K^{[l]}\}$ equally and sequentially into
     sub-modules $Z \leftarrow \{z^{[1]}, z^{[2]}, \dots, z^{[n_l]}\}$ ;
10 for $l \leftarrow 1$ to $L$ by 1 do
11     split and merge $z^{[l]}z^{[l+1]}$ into $z_1^{[l,l+1]}, z_2^{[l,l+1]}, \dots z_{n_l}^{[l,l+1]}$;
12     set $e$ to {} ;
13     for $i \leftarrow 1$ to $n_l$ by 1 do
14         allocate and compute $e_i = d_i \leftarrow z_i^{[l,l+1]}$ ;
15         $e = e \leftarrow e_i$;
16     end
17     $z^{[l+2]} \leftarrow e$;
18 end
19 return $Z$
```

3.3. Forward pass and backward pass parallelisation

To further improve DNN training performance, we address the update locking problem by parallelising the forward pass and backward pass, and develop a cost model to analyse the total execution cost.

3.3.1. Forward pass and backward pass parallelisation

Here, we parallelise forward pass, backward pass and parameter synchronisation processes to address the update locking problem caused by the sequential execution of forward pass and backward pass computations. Fig. 3 represents model parallelism with non-parallel and parallel forward and backward pass computations. As shown in Fig. 3(a), the forward pass computes the activation of modules F_i from $i = 1$ to n and the backward pass calculates the error gradients B_i in reverse order from output layer to input layer, that is $i = n$ to $i = 1$ (as in Eq. (3)), sequentially at iteration t. To reduce dependencies among the layers for both activation and gradient calculations, we parallelise the forward and backward passes by splitting the dataset by the number of available GPUs such that $X = \{x_1, x_2, \dots, x_n\}$; each data batch is divided into minibatches of size b/n. Each sub-module z_i, where $i = \{1, 2, \dots, n\}$ is fed with different dataset chunks. Each sub-module computation runs on different GPUs concurrently to ensure better forward and backward pass throughput as well as hardware utilisation efficiency. At each iteration, we first perform input and output computations of a module for the forward and backward pass before transferring dependencies between the two propagations. $B_{i,j}$ represents the backward pass of sub-module i and micro-batch j while $F_{i,j}$ represents the forward pass of module i and micro-batch j. For instance, assume there are four available


Stage3 / GPU3
Stage2 / GPU2
Stage1 / GPU1
Stage0 / GPU0
F0 F1 F2 F3 B3 B2 B1 B0 F0 F1 F2 F3 B3 B2 B1 B0
F0,0 F0,1 F1,0 F2,0 F3,0 B3,0 F3,1 B3,1 F3,2 B3,2 F3,3 B3,3
B2,0 F2,1 B2,1 F2,2 B2,2 F2,3 B2,3
B1,0 F1,1 B1,1 F1,2 B1,2 F1,3 B1,3
B0,0 F0,2 B0,1 F0,3 B0,2 B0,3

(a) Model parallelism with non-parallel forward and backward pass computations

(b) Model parallelism with parallel forward and backward pass computations

Fig. 3. Forward and backward pass computations. The left figure is model parallelism with non-parallel forward and backward pass computations; a single data batch across GPUs leads to several unutilised hardware. The figure on the right shows parallel forward and backward pass computations with mini-batches for all GPUs.

GPUs (i.e. n = 4), then there will be four sub-modules, z_i, where $i = \{1, 2, \ldots, 4\}$ as described in Algorithm 1 and four micro-batches j such that $j = \{1, 2, \ldots, 4\}$. The forward pass F_1, F_2, F_3, F_4 and backward pass B_1, B_2, B_3, B_4 in Fig. 3(a) now become $F_{0,0}$, $F_{0,1}$, $F_{0,2}$, $F_{0,3}$, $F_{1,0}$, $F_{1,1}$, $F_{1,2}$, $F_{1,3}$, $F_{2,0}$, $F_{2,1}$, $F_{2,2}$, $F_{2,3}$, $F_{3,0}$, $F_{3,1}$, $F_{3,2}$, $F_{3,2}$ for forward pass and $B_{0,0}$, $B_{0,1}$, $B_{0,2}$, $B_{0,3}$, $B_{1,0}$, $B_{1,1}$, $B_{1,2}$, $B_{1,3}$, $B_{2,0}$, $B_{2,1}$, $B_{2,2}$, $B_{2,3}$, $B_{3,0}$, $B_{3,1}$, $B_{3,2}$, $B_{3,2}$ for backward pass. To reduce the idleness of devices occurring in Fig. 3(a) and efficiently utilise devices' memories, the computations of the partitioned forward pass and backward pass are arranged randomly and processed on the different GPUs simultaneously, as shown in Fig. 3(b). However, the computations of $B_{0,0}$ depends on the output of $F_{0,0}$, $B_{0,1}$ depends on the output of $F_{0,1}$, $B_{0,2}$ depends on the output of $F_{0,2}$ and $B_{0,3}$ depends on the output of $F_{0,3}$ in stage 0. Similar operations are performed in stages 1, 2, and 3.

To improve training performance, asynchronous parameter update and gradient accumulation methods are used to perform parameter updates among the module with all training batches rather than a single batch. At the training process's start, the first batch training samples are used for forward pass calculation, and backward pass starts with the same batch immediately forward pass finishes. Then the forward and backward pass computations perform for the subsequent batches. The training process continuously feeds new training batches for forward and backward pass calculation tasks with different GPUs to ensure throughput and system utilisation.

3.3.2. Cost model

We formulate a cost model to analyse the computation time performance of the forward pass and backward pass parallelisation method. First, we define the time taken to perform the forward and backward passes for module i using micro-batch j as $T(F_{i,j})$ and $T(B_{i,j})$ respectively. We assume that these times are constant such that

$$T(F_{i,j}) = T_F \text{ and } T(B_{i,j}) = T_B \;\forall\; (i, j). \tag{10}$$

The communication time between GPUs to transfer model weights is taken to be a constant T_C. We can now write the cost model for the computation time for the two cases corresponding to Figs. 3(a) and 3(b).

For the case without overlapping of forward and backward passes, depicted in Fig. 3(a), we can write the time to complete a mini-batch with non-overlapped computation as

$$T_{NO} = n^2(T_F + T_B + 2T_C), \tag{11}$$

since each micro-batch/module combination requires a time $T_F + T_B$ for its processing, in each case there is a communication cost of T_C incurred, and there are n microbatches over n modules.

For the case with overlapping forward and backward passes, we note that the initial stage in which the first microbatch is processed, which is represented by the initial diagonal ramp in Fig. 3(b), requires a time $n(T_F + T_c)$. Similarly the final stage in which the backward pass is performed on microbatch $n - 1$ requires $n(T_B + T_c)$. In between these times, the processing of module $n - 1$ is the limiting process; this GPU will alternate between performing forward and backward passes on the micro-batches in turn. While this is happening, the other processes may be performing either forward or backward passes. The time taken for this phase of the computation is therefore $2(n-1)\max(T_F, T_B)$ since within each of the $2(n - 1)$ time slices in the diagram corresponding to this phase, there are both forward and backward passes being processed. Note there is no communication between GPUs in this phase, so there is no term in T_C. Combining the terms, we now write the total time taken for the overlapped computation as

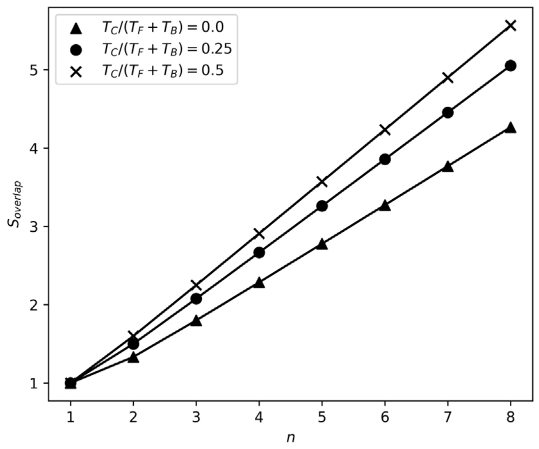


Fig. 4. Speedup due to computation overlap from the cost model presented in Eq. (13), for three different values of the ratio of communication time to compute time, with $T_F = T_B$.

$$T_O = n(T_F + T_B + 2T_C) + 2(n - 1)\max(T_F, T_B). \tag{12}$$

We can now write the parallelisation speedup derived from the overlapping process $S_{\text{overlap}} = T_{NO}/T_O$, as

$$S_{\text{overlap}} = \frac{n^2(T_F + T_B + 2T_C)}{n(T_F + T_B + 2T_C) + 2(n - 1)\max(T_F, T_B)}. \tag{13}$$

This quantity is plotted for different values of the ratio of communication time to compute time, assuming that $T_F = T_B$ in Fig. 4.

We can also compare the contributions to these times due to communication only, which are $2n^2T_C$ for case (a) and $2nT_c$ for case (b). The proposed overlapping method therefore reduces the communication cost by a factor n.

4. Experimental evaluation

In this section, we carry out the experiments on the proposed method. Section 4.1 describes the use cases; DNN models and datasets used. In Section 4.2, we provide definitions of evaluation metrics; speedup, accuracy and training time. Section 4.3 provides the hardware and software settings of the experiments. In

Section 4.4, we discuss the experimental results and compare the performance with the existing parallel methods. The goals of the experiments are to implement the proposed method as described in Section 3 and evaluate, through realistic use cases, its scalability and performance in a multi-GPU environment. Moreover, we also compare the speedup and accuracy performance with some state-of-the-art parallel methods including DDG [35], FDG [11], DSP [36], and GABRA [31].

4.1. Use case description

To provide a robust evaluation of our proposed approach, we select two use cases including: (1) A real-world application in relation to Alzheimer's disease diagnosis based on our original proposed model (3D-ResAttNet). The original model has been effective for disease diagnosis. However, it is a non-parallel model. Given the large size of sMRI images, it is necessary to accelerate the computing performance. Hence parallelisation is required; (2) Classification tasks with ResNets and VGG 16 using CIFAR-10 and CIFAR-100 datasets (These models and datasets are widely used by many researchers for benchmarking evaluation). The rationale behind these selection are mainly dependent on the data size and model structure complexity. CIFAR-10 and CIFAR-100 datasets are two dimensional with relatively small size while the sMRI data is three dimensional, and is a large dataset. In addition, our proposed model is 3D ResNet with attention layer, which is more complex than ResNets and VGG 16, which allows for evaluating the proposed method with more complex deep learning models with a range of layer types.

4.1.1. Use case 1: 3D-ResAttNet for Alzheimer's disease

We apply the proposed method on our previous non-parallel 3D-ResAttNet for automatic detection of the progression of AD and its Mild Cognitive Impairments (MCIs) - Normal cohort (NC), Progressive MCI (pMCI), and Stable MCI (sMCI) from sMRI scans [40]. The network consists of 3D Conv blocks, Residual self-attention blocks, and Explainable blocks. 3D convolutions exploit a 3D filter to calculate the low-level feature representations of the output shape as a 3-dimensional volume space. The residual self-attention block combines two important network layers: the residual network layer and the Self-attention layer.

The residual network layer comprises two Conv3D blocks consisting of 3 × 3 3D convolution layers, 3D batch normalisation and Rectified Linear Unit (ReLU). The explainable block uses 3D Grad-CAM to improve the model decision. We adopt the dataset from the Alzheimer's Disease Neuroimaging Initiative (ADNI) database (http://adni.loni.usc.edu) as the benchmark for the performance evaluation. The dataset has four classes of MRI scans images, developed in 2003 by Dr Michael W. Weiner under the public–private partnership. As shown in Table 2, it contains 1193 MRI scans 389 Alzheimer's Disease (AD), 400 Normal Cohort (NC), 232 Stable Mild Cognitive Impairment (sMCI) and 172 Progressive Mild Cognitive Impairment (pMCI) patients.

4.1.2. Use case 2: ResNets and VGG16 for classification of CIFAR-10 and CIFAR-100 datasets

We also apply ResNet18 [41], ResNet34 [42], ResNet50 [43] and VGG16 models [44] for the classification of the CIFAR-10 and CIFAR-100 images [45] to further evaluate the robustness of our proposed method. The ResNet networks consist of 2D convolutional layers with 3 × 3 filters, batch normalisation, rectified linear unit and residual block layers, ending with an average pooling layer and a fully-connected layer. The VGG 16 consists of convolutional layers, Max Pooling layers, and Dense layers. ResNet and VGG model structures are simpler than the 3D-ResAttNet model [40]. The CIFAR-10 and CIFAR-100 datasets are commonly used for benchmarking DNN models. The CIFAR-10 dataset contains 60,000 images with 32 × 32 pixels, divided into 10000 test images and 50000 training images. The images are classified into ten classes - aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck; each has 6000 images. The CIFAR-100 dataset contains 100 classes each containing 600 images (500 training and 100 testing images). The CIFAR-100 classes are further grouped into 20 superclasses, and each image can be identified by class and superclass labels.

Table 2
ADNI database descriptions.

Class	Number/Size	Gender (Male/Female)	Age (Mean/Std)	MMSE (Mean/Std)
AD	389/1.4 GB	202/187	75.95/7.53	23.28/2.03
pMCI	172/484 MB	105/67	75.57/7.13	26.59/1.71
sMCI	232/649 MB	155/77	75.71/7.87	27.27/1.78
NC	400/2.4 GB	202/198	76.02/5.18	29.10/1.01

4.2. Experimental evaluation metrics

We adopted Speedup (S), Accuracy (ACC) and Training Time (TT) for performance evaluation of out proposed method. S measures the scalability and computing performance and defined as:

$$S = T_s/T_p \tag{14}$$

T_s denotes the computing time on a single GPU, i.e. the total runtime for training process from the beginning to the end including the layer partitioning, merging, forward and backup pass operations and other time costs. T_p indicates the computing time on p GPUs (the total runtime for training process from the beginning to the end). ACC measures the classification accuracy and is defined as:

$$ACC = (TP + TN)/(TP + TN + FP + FN) \tag{15}$$

where TP = True positives, FP = False positives, TN = True negatives and FN = False negatives. TT measures time taken for training of the DNN models using the proposed approach and other existing parallel methods

4.3. System configuration

Hardware: Our experiments are conducted on an Amazon Web Service (AWS) EC2 p3.16xlarge instance. The p3.16xlarge instance consists of 8 NVIDIA Tesla V100 GPUs with NVLink technology, 128 GB GPU memory, 64 vCPUs, 4488 GB memory, and 25 Gbps network bandwidth.

Software: we exploit the following software configuration and installation: Ubuntu 18.04 as the backbone for other software installation, Python 3.7.3, PyTorch 1.2.0 as the deep learning framework [46], Torchvision 0.4.0, Numpy 1.15.4, Tensorboardx 1.4, Matplotlib 3.0.1, Tqdm 4.39.0, nibabel, fastai, and NVIDIA Collective Communications Library (NCCL) CUDA toolkit 10.2 - a library of multi-GPU collective communication primitives [47].

4.4. Experiments results and discussions

We have investigated the accuracy and speedup performances of the proposed method on the use case 1, 3D-ResAttNets for two classification tasks: sMCI vs pMCI and AD vs NC, and the use case 2, ResNets and VGG16 on CIFAR-10 and CIFAR-100 with varying numbers of GPUs (ranging from 1 to 8). Furthermore, we have also compared performances of the proposed against GABRA [31], our previous parallelisation method, and other three state-of-art methods, including DDG [35], DSP [36] and FDG [11]. We have

Table 3
Use case 1: training performances 3D ResAttNet18 and 3D ResAttNet34 with ADNI dataset using the proposed method.

#GPUs	3D-ResAttNet18				3D-ResAttNet34			
	sMCI vs. pMCI		AD vs. NC		sMCI vs. pMCI		AD vs. NC	
	ACC	TT (min)	ACC	TT (min)	ACC	TT (min)	ACC	TT (min)
1	0.79	29	0.93	55	0.81	34	0.94	61
2	0.80	11	0.92	22	0.81	14	0.94	24
3	0.81	9	0.92	20	0.82	12	0.95	21
4	0.80	7	0.93	15	0.83	11	0.93	17
5	0.80	6	0.94	13	0.81	9	0.93	15
6	0.79	5	0.93	11	0.82	7	0.95	12
7	0.79	4	0.93	8	0.83	5	0.94	10
8	0.80	3	0.93	7	0.82	4	0.94	8

Table 4
Use case 2: training performances ResNet18, ResNet34, ResNet50 and VGG16 with CIFAR-100 using the proposed method.

#GPUs	ResNet18		ResNet34		ResNet50		VGG16	
	ACC	TT (min)	ACC	TT (min)	ACC	TT (min))	ACC	TT (min)
1	0.94	26	0.94	34	0.94	41	0.93	44
2	0.93	14	0.94	19	0.94	23	0.93	25
3	0.94	13	0.94	16	0.94	20	0.93	22
4	0.94	11	0.93	15	0.94	16	0.94	18
5	0.93	10	0.94	13	0.93	14	0.93	15
6	0.94	8	0.93	10	0.94	12	0.93	14
7	0.94	6	0.93	8	0.94	9	0.93	12
8	0.93	4	0.94	6	0.94	8	0.93	10

Table 5
Use case 2: training performances ResNet18 and ResNet34, ResNet50 and VGG16 with CIFAR-10 using the proposed method.

#GPUs	ResNet18		ResNet34		ResNet50		VGG16	
	ACC	TT (min)	ACC	TT (min)	ACC	TT (min)	ACC	TT (min)
1	0.93	25	0.94	32	0.93	39	0.93	42
2	0.94	12	0.94	18	0.94	22	0.94	24
3	0.94	11	0.93	15	0.94	18	0.93	20
4	0.93	10	0.94	14	0.93	15	0.93	17
5	0.94	9	0.94	11	0.93	12	0.94	13
6	0.93	7	0.94	10	0.94	11	0.93	12
7	0.93	5	0.94	7	0.93	8	0.93	10
8	0.94	3	0.94	5	0.93	6	0.94	8

used Rectified linear unit(Relu) as the activation function and optimised model parameters with SGD, a stochastic optimisation algorithm. In addition, we set other training parameters, including a batch size of six samples, cross-entropy as the loss function, and 50 epochs for better convergence. We set initial learning rate (LR) as 1×10^{-4}, then reduced by 1×10^{-2} with increased iterations.

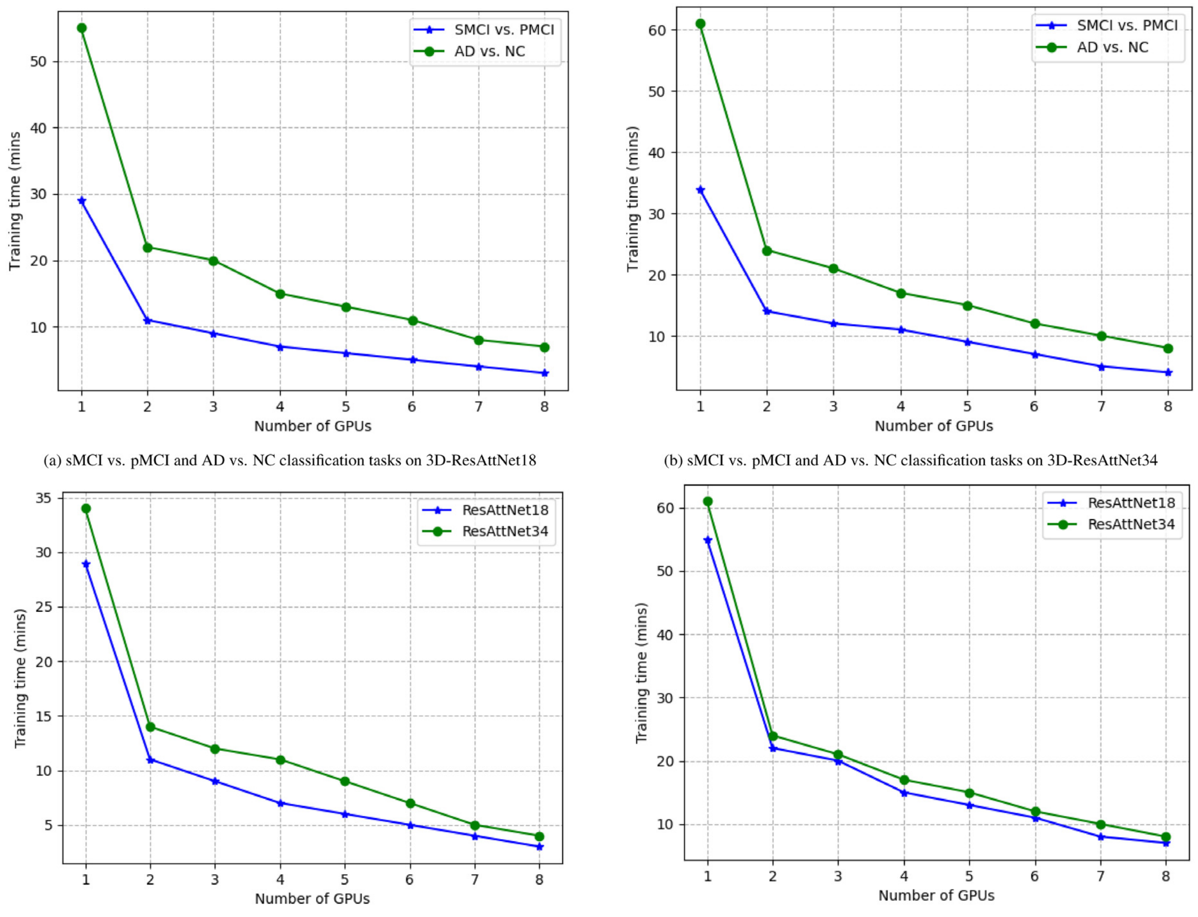
4.4.1. Training time

Table 3 shows the training results of use case 1, 3D-ResAttNets (3D-ResAttNet18 and 3D-ResAttNet34) on the two classification tasks: sMCI vs pMCI and AD vs NC. Tables 4 and 5 show the training results of use case 2, ResNets and VGG16 with CIFAR-10 and CIFAR-100 using the proposed parallelisation method. In addition, to further demonstrate the effectiveness of the proposed method, we have conducted experiments to show the time taken for merging and splitting, as shown in Table 6, the durations (communication and computing) are insignificant and have little or no effect on the training time.

Figs. 5 and 6 visualise the performance of use cases 1 and 2 using the proposed parallelisation method in terms of the training time with the varying number of GPUs, respectively. Both show that as the number of GPUs increases, the training time decreases. For instance, in the use case 1, the sMCI vs pMCI classification task on 3D-ResAttNet18 gives 29 min when using a single GPU, 11 min with 2 GPUs, and 3 min with 8 GPUs. The same performance trend is seen for AD vs NC classification tasks on 3D-ResAttNet18 and sMCI vs pMCI and AD vs NC classification tasks on 3D-ResAttNet34. Likewise, in the use case 2, the training of ResNet18 with CIFAR-100 images gives 26 min with a single GPU, 14 min with 2 GPUs, and 13 min with 3 GPUs. ResNet34 with CIFAR-100 dataset gives 34 min on a single GPU, 19 min with 2 GPUs, and 16 min with 3 GPUs. ResNet50 with CIFAR-100 dataset gives 41 min on a single GPU, 23 min with 2 GPUs, 20 min with 3 GPUs, and so on. Also, VGG16 with CIFAR-100 dataset gives 44 min on a single GPU, 25 min with 2 GPUs, 22 min with 3 GPUs, and so on. As shown in Fig. 6, in all cases of ResNet 18, 34 and 50 and VGG 16, the training time decreases as the number of GPUs increases. The ResNet 50 takes more training time than the ones of ResNet 18 and 34 in all cases. Since the VGG model is more complex than ResNets, it has the longest training time. This indicates that the more the complex model is, the longer the training time.

Table 6
Splitting and merging times measured for two representative cases. The times are orders of magnitude smaller than the training times in both cases.

#GPUs	Splitting and merging time (s)	
	Case 1 (3D-ResAttNet18)	Case 2 (ResNet50 with CIFAR-100)
1	–	–
2	0.0002548	0.0002382
3	0.0002602	0.0002410
4	0.0002814	0.0002604
5	0.0002879	0.0002697
6	0.0002915	0.0002741
7	0.0002983	0.0002815
8	0.0003011	0.0002879



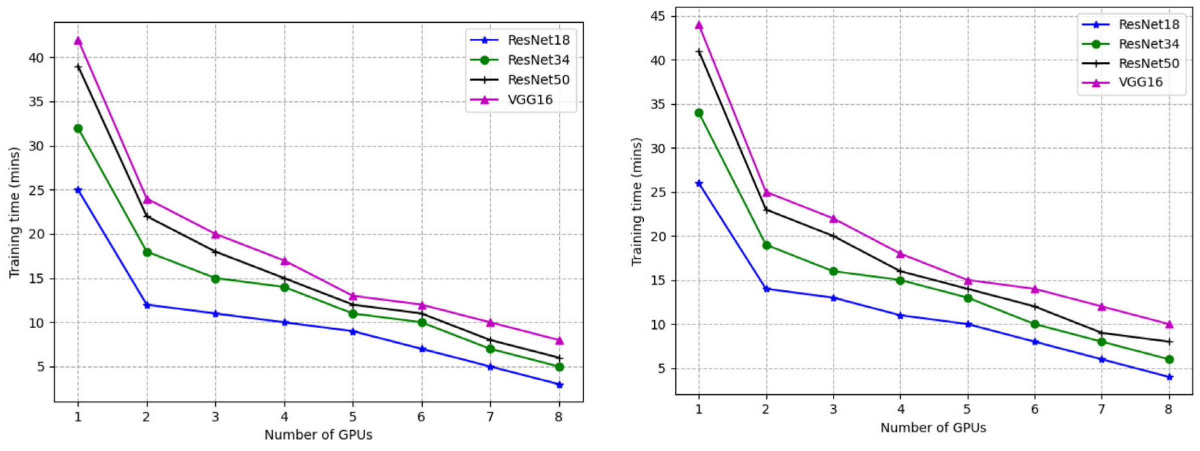
(a) sMCI vs. pMCI and AD vs. NC classification tasks on 3D-ResAttNet18

(b) sMCI vs. pMCI and AD vs. NC classification tasks on 3D-ResAttNet34

(c) 3D-ResAttNet18 vs. 3D-ResAttNet34 for sMCI vs. pMCI classification tasks

(d) 3D-ResAttNet18 vs. 3D-ResAttNet34 for AD vs. NC classification tasks

Fig. 5. Use case 1: training time performance of 3D-ResAttNet18 and 3D-ResAttNet34 using the proposed method .



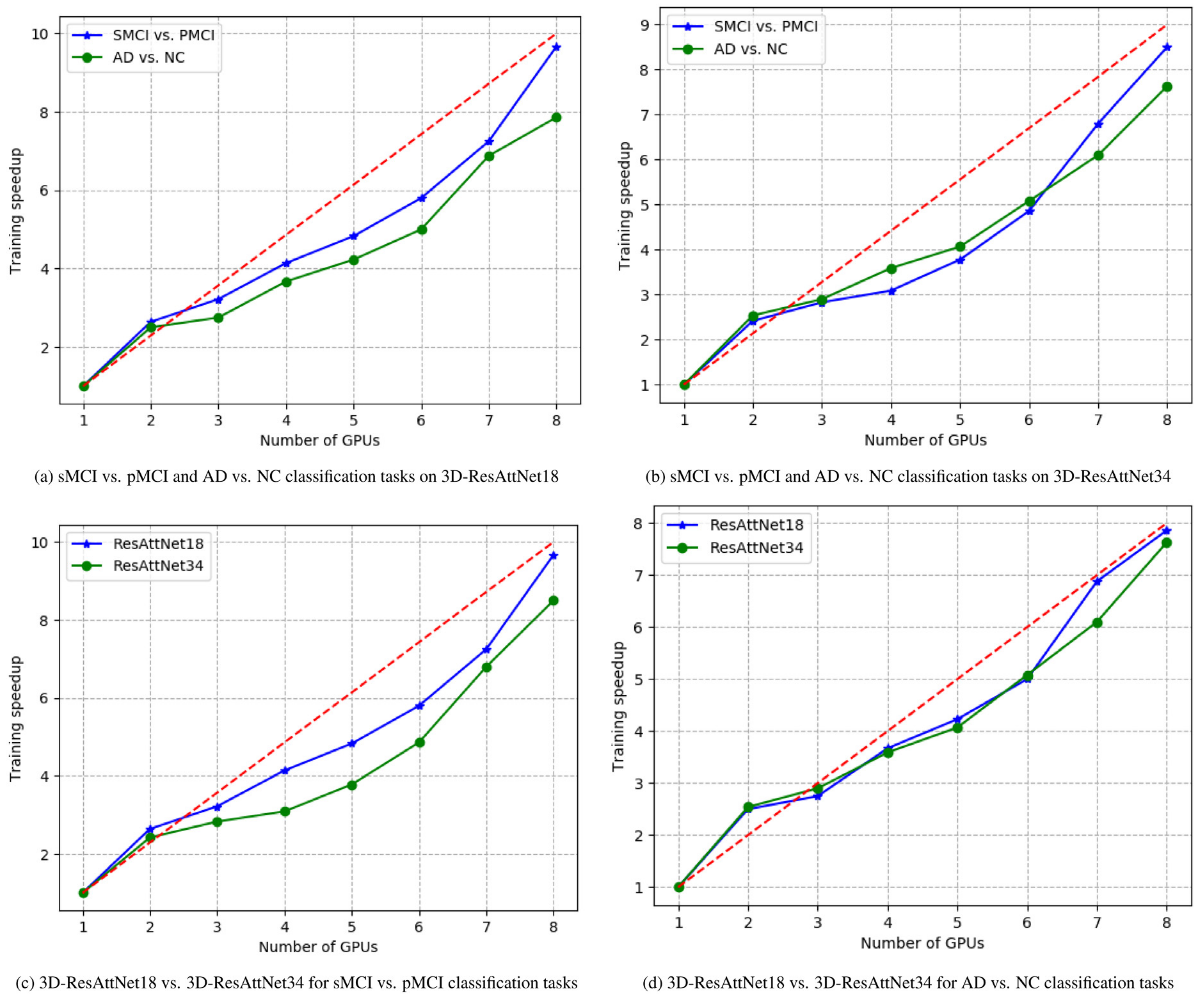
(a) ResNet18, ResNet34, ResNet50 and VGG16 wih CIFAR-10

(b) ResNet18, ResNet34, ResNet50 and VGG16 wih CIFAR-100

Fig. 6. Use case 2:Training time performance of ResNet18, ResNet34, ResNet50 and VGG16 using the proposed method.

4.4.2. Speedup

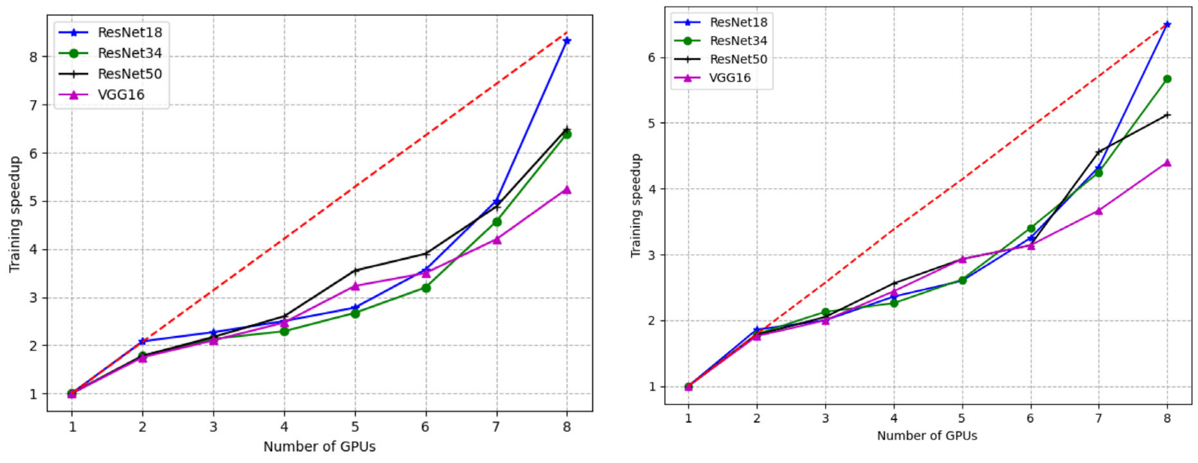
We investigate the relationship between speedup and the number of GPUs. The speedup (S) measures the scalability and computing performance and is defined as in Eq. (14). Figs. 7 (c) and (d) show, for use case 1, the speedup performance of ResAttNets using the proposed method calculated based on the training time with varying numbers of GPUs. The figure shows that the speedup increases almost linearly with the number of GPUs, which illustrates the scalability of the proposed method. Specifically, for AD vs. NC classification task with 3D-ResAttNet18, the training speedup on 1, 2, 3, 4, 5, 6, 7, and 8 GPUs are 1, 2.5, 2.75, 3.67, 4.23, 5, 6.88 and 7.86 respectively. The sMCI vs pMCI



(a) sMCI vs. pMCI and AD vs. NC classification tasks on 3D-ResAttNet18 (b) sMCI vs. pMCI and AD vs. NC classification tasks on 3D-ResAttNet34

(c) 3D-ResAttNet18 vs. 3D-ResAttNet34 for sMCI vs. pMCI classification tasks (d) 3D-ResAttNet18 vs. 3D-ResAttNet34 for AD vs. NC classification tasks

Fig. 7. Use case 1: speedup performance of 3D-ResAttNets using the proposed method.

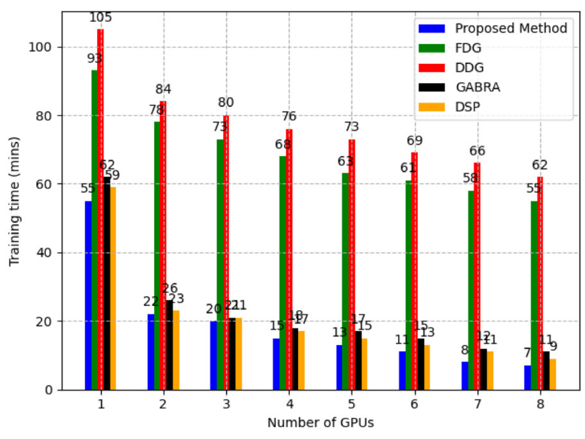


(a) ResNet18, ResNet34, ResNet50 and VGG16 wih CIFAR-10 (b) ResNet18, ResNet34, ResNet50 and VGG16 wih CIFAR-100

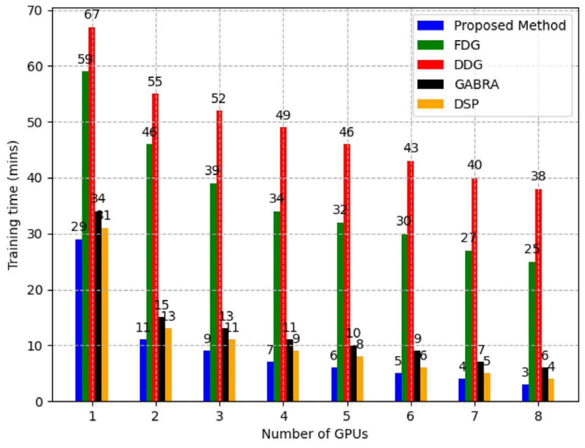
Fig. 8. Use case 2: speedup performance of ResNet18, ResNet34, ResNet50 and VGG16 using the proposed method with CIFAR datasets.

classification task with 3D-ResAttNet18, the training speedup for 1, 2, 3, 4, 5, 6, 7, and 8 GPUs are 1, 2.64, 3.22, 4.14, 4.83, 5.8, 7.25 and 9.67 respectively. A similar speedup performance trend was also observed in the sMCI vs pMCI and AD vs NC classification tasks with 3D-ResAttNet34.

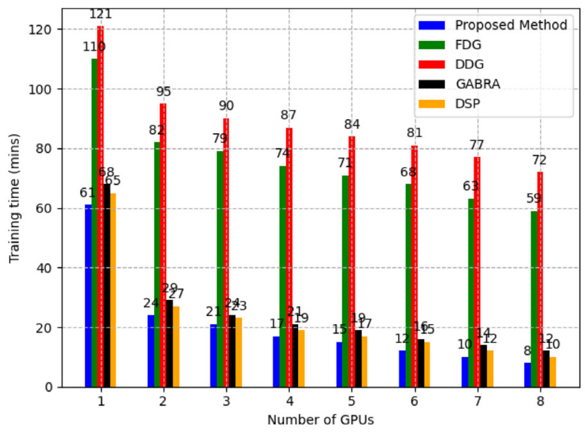
Also, Fig. 8 shows, for the use case 2, the speedup performance of ResNets — ResNet18, ResNet34 and ResNet50, and VGG16 using the proposed method. In Fig. 8(b), for ResNet18 with CIFAR-100, the training speedup for 1, 2, 3, 4, 5, 6, 7, and 8 GPUs are 1, 1.86, 2, 2.36, 2.6, 3.25, 4.33 and 6.5, respectively, while that of



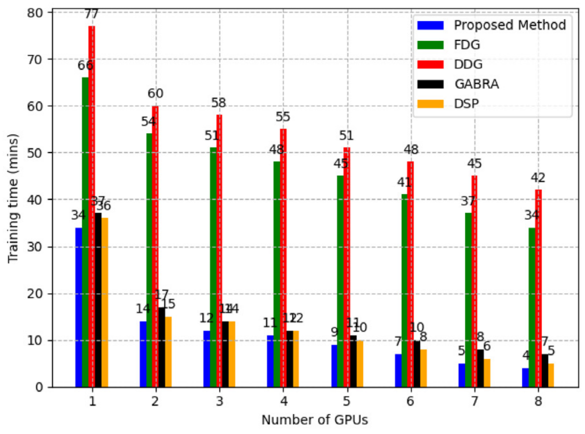
(a) Training Time of 3D-ResAttNet18 over AD vs. NC Dataset



(b) Training Time of 3D-ResAttNet18 over sMCI vs. pMCI Dataset



(c) Training Time of 3D-ResAttNet34 over AD vs. NC Dataset



(d) Training Time of 3D-ResAttNet34 over sMCI vs. pMCI Dataset

Fig. 9. Use case 1: training time of 3D-ResAttNet18 and 3D-ResAttNet34 using the proposed method, FDG, DDG, DSP and GABRA.

ResNet34 with CIFAR-100 are 1, 1.79, 2.13, 2.26, 2.62, 3.4, 4.25 and 5.67, respectively. ResNet50 with CIFAR-100 gives speedup of 1, 1.78, 2.05, 2.56, 2.93, 3.14, 4.56 and 5.12 for 1, 2, 3, 4, 5, 6, 7, and 8 GPUs, respectively. Similarly, VGG16 with CIFAR-100 gives speedups of 1, 1.76, 2.0, 2.44, 2.93, 3.14, 3.67 and 4.4 for 1, 2, 3, 4, 5, 6, 7, and 8 GPUs, respectively. The same speed performance trends are also shown in Fig. 8(a) for ResNet18, ResNet34, ResNet50 and VGG16 with CIFAR-10. In all cases, the speedup increases as the number of GPUs increases, which indicates that the proposed method is scalable.

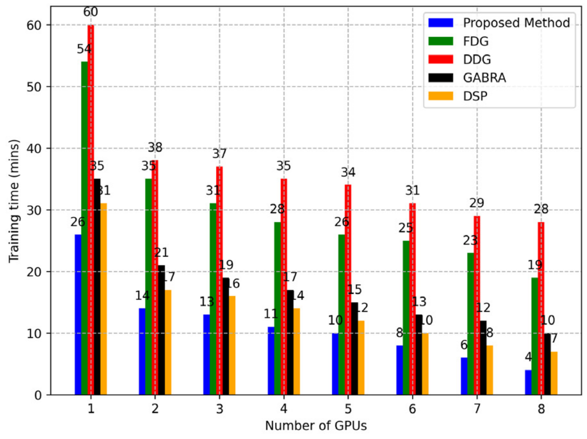
Although the speedup in all cases increases with the number of GPUs, within this broad behaviour there are some differences between models and datasets. In some cases, we observe superlinear speedup, with parallel efficiencies greater than 1 (ResAttNet18 and ResAttNet34 on the SMCI vs. PMCI task). This may be explained by lower than expected performance in the single GPU case. One possible reason for this is the size of a model which may cause issues on the hardware which do not manifest when the model is split between GPUs, such as frequent cache misses. This would result in an artificially high value for the training time on one GPU, leading to higher apparent speedup when more GPUs are used. This effect only manifests with the larger data and models, consistent with the hypothesis. In many cases we see a performance below the linear trend which then improves beyond around 4–5 GPUs. There is some indication of this in the performance model developed in Section 3.3.2, in which the curves are not linear and dip slightly below the linear relationship, although the effect observed experimentally is more pronounced. This may also be due to the superlinear scaling effect of reducing memory usage on individual GPUs; the scaling initially follows a line similar to the cost model, with some increase in speedup for larger numbers of GPUs, as smaller per-GPU models are able to make more efficient use of cache memory.

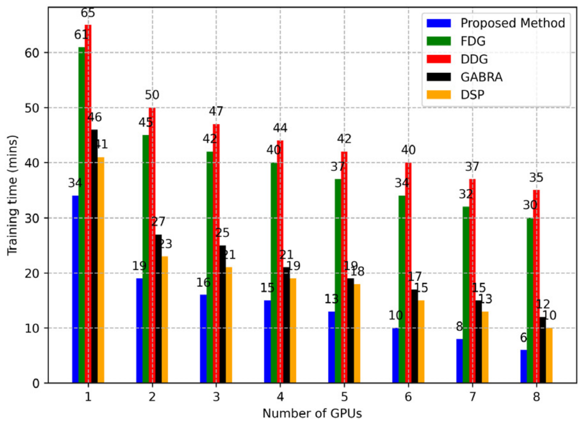
4.4.3. Accuracy

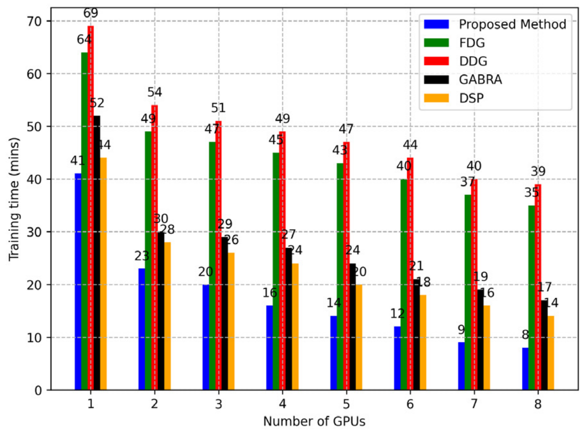
As shown in Tables 3 and 4, the accuracy performance of our proposed model on two use cases maintains the same accuracy level with different numbers of GPUs, comparing to the non-parallel approach. Specifically, there is no correlation between the test accuracy and the number of GPUs. For instance, in the use case 1, our proposed method using 3D-ResAttNet34 provides test accuracies: 0.94, 0.94, 0.95, 0.93, 0.93, 0.95, 0.94 and 0.94 on 1, 2, 3, 4, 5, 6, 7, and 8 GPUs respectively. The same behaviours are also shown in the accuracy performance of 3D-ResAttNet18 and the use case 2, ResNet18, ResNet34, ResNet50 and VGG16 with the different numbers of GPUs.

4.4.4. Comparison of the proposed method and existing parallel methods

We have compared the proposed method with our previous parallelisation method, GABRA and other three state-of-art methods: DDG, DSP and FDG. Figs. 9 and 10 show the experiment results of use cases 1 and 2, respectively. The proposed method outperforms the GABRA, DDG, DSP and FDG in terms of training time. For instance, in use case 1, for 3D ResAttNet18 on AD


(a) ResNet18


(b) ResNet34


(c) ResNet50

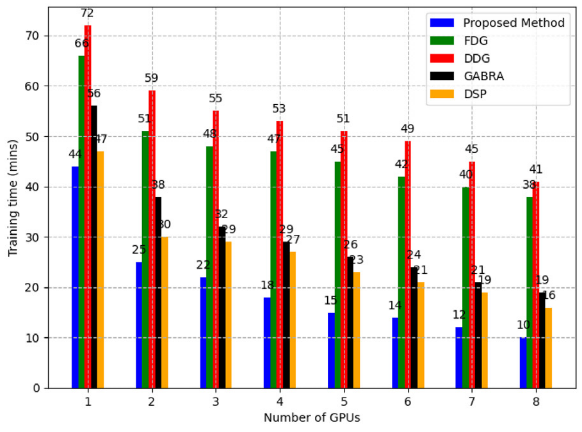

(d) VGG16

Fig. 10. Use case 2: training time of ResNet18, ResNet34, ResNet50 and VGG16 with CIFAR-100 using the proposed method, FDG, DDG, DSP and GABRA.

Table 7
Memory usage comparison between the existing parallel methods and our proposed method. The proposed method took an average of 66% and 58% of the memory to train 3D-ResAttNet34 and ResNet50 models, respectively and outperforms the GABRA, DDG, DSP and FDG.

Methods	3D-ResAttNet34 (AD vs. NC)	ResNet50 (CIFAR-100)
Proposed method	66%	58%
FDG	78%	73%
DDG	89%	81%
DSP	69%	62%
GABRA	73%	66%

vs NC and sMCI vs pMCI classification tasks, the training time incurred by the proposed method is lower than the ones of GABRA, DDG, DSP and FDG. Similarly, there are related trends when comparing the proposed method with the GABRA, DDG, DSP and FDG for the distributed training of 3D-ResAttNet34 for two classification tasks: AD vs NC and sMCI vs pMCI, for instance, In Fig. 9(c), when training 3D-ResAttNet34 over AD vs NC on two GPUs, the performance improvements of our proposed model over the existing methods including FDG, DDG, GABRA and DSP achieves 70%, 74.73%, 17.24% and 11.11% respectively, which demonstrates the effectiveness of our proposed method. The same performance trend also recorded in the use case 2, ResNet18, ResNet34, ResNet50 and VGG16 with CIFAR-100 dataset. In addition, Table 7 shows the comparison of the memory usage of the exiting methods and our proposed method. Our proposed method consumed an average of 66% and 58% of the system memory to train 3D-ResAttNet34 and ResNet34 models, respectively. In all results, our proposed method outperforms the existing methods, FDG DDG, DSP and GABRA, in terms of memory usage as well.

5. Conclusion and future works

In this paper, we have proposed layer-wise partitioning and merging to solve forward and backward locking problems by performing computations of network layers across multiple available devices rather than a single device. We have also proposed a forward pass and backward pass parallelisation method to address the update locking problem associated with the sequential execution of forward pass and backward pass computations. We applied the proposed method to train two CNNs — our previous 3D Residual Attention Deep Neural Network (3D-ResAttNet) model on real-world Alzheimer's Disease (AD) datasets and the ResNet and VGG models on CIFAR-10 and CIFAR-100 datasets. The experimental results show that the proposed method achieves almost linear speedup without compromising accuracy performance, demonstrating its scalability and efficient computing capability. In addition, the comparison evaluation shows that the proposed method can earn a considerable speedup and reduced memory consumption when training the deep feed-forward neural network and outperforms the state-of-art methods on the same benchmark datasets. The proposed method ensures a well-balanced computing workload across multiple GPUs to achieve the best possible runtime performance while maintaining the original network accuracy and has the potential to work effectively in a multi-node environment. Future work will evaluate the proposed model on more DNN models and across multiple nodes, each with a certain number of GPUs in a distributed setting.

CRediT authorship contribution statement

S.B. Akintoye: Conceptualization, Methodology, Data acquisition and processing, Analysis, Writing – original draft. **L. Han:** Conceptualization, Methodology, Data acquisition and processing, Analysis, Writing – review & editing, Project administration, Funding acquisition. **H. Lloyd:** Conceptualization, Data acquisition and processing, Analysis, Writing – review & editing. **X. Zhang:** Conceptualization, Data acquisition and processing, Analysis, Writing – review & editing. **D. Dancey:** Conceptualization, Analysis, Writing – review & editing. **H. Chen:** Conceptualization, Analysis, Writing – review & editing. **D. Zhang:** Conceptualization, Analysis, Writing – review & editing, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgement

The work reported in this paper has formed part of the project by Royal Society - Academy of Medical Sciences Newton Advanced Fellowship (NAF \R1\180371).

References

[1] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 2818–2826.

[2] J. Hu, L. Shen, G. Sun, Squeeze-and-excitation networks, in: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 7132–7141.

[3] Y. Liu, A. Xu, Z. Chen, Map-based deep imitation learning for obstacle avoidance, in: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, 2018, pp. 8644–8649, http://dx.doi.org/10.1109/IROS.2018.8593683.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., TensorFlow: A system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI '16, USENIX Association, USA, 2016, pp. 265–283.

[5] J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M. Ran-zato, A. Senior, P. Tucker, K. Yang, A.Y. Ng., Large scale distributed deep networks, in: Proceedings of NIPS, 2012, pp. 1232–1240.

[6] N. Dryden, N. Maruyama, T. Moon, T. Benson, M. Snir, B. Van Essen, Channel and filter parallelism for large-scale CNN training, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, 2019.

[7] Y. Cheng, Derivation of the backpropagation algorithm based on derivative amplification coefficients, 2021, arXiv abs/2102.04320.

[8] F. Fan, W. Cong, G. Wang, General backpropagation algorithm for training second-order neural networks, Int. J. Numer. Methods Biomed. Eng. 34 5 (5) (2018) e2956.

[9] M. Jaderberg, W.M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, K. Kavukcuoglu, Decoupled neural interfaces using synthetic gradients, in: Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML '17, JMLR.org, 2017, pp. 1627–1635.

[10] Z. Huo, B. Gu, Q. Yang, H. Huang, Decoupled parallel backpropagation with convergence guarantee, in: ICML, 2018.

[11] H. Zhuang, Y. Wang, Q. Liu, Z. Lin, Fully decoupled neural network learning using delayed gradients, IEEE Trans. Neural Netw. Learn. Syst. PP (2021).

[12] Z. Huo, B. Gu, H. Huang, Training neural networks using features replay, 2018, arXiv abs/1807.04511.

[13] X. Liu, C. Wu, M. Menta, L. Herranz, B. Raducanu, A.D. Bagdanov, S. Jui, J. van de Weijer, Generative feature replay for class-incremental learning, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, CVPRW, 2020, pp. 915–924.

[14] Z. Jia, S. Lin, C. Qi, A. Aiken, Exploring hidden dimensions in parallelizing convolutional neural networks, 2018, arXiv abs/1802.04924.

[15] Z. Jia, M. Zaharia, A. Aiken, Beyond data and model parallelism for deep neural networks, 2019, arXiv abs/1807.05358.

[16] F. Seide, H. Fu, J. Droppo, G. Li, D. Yu, On parallelizability of stochastic gradient descent for speech DNNS, in: 2014 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, 2014, pp. 235–239, http://dx.doi.org/10.1109/ICASSP.2014.6853593.

[17] A. Krizhevsky, One weird trick for parallelizing convolutional neural networks, 2014, arXiv abs/1404.5997.

[18] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J.R. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, J. Dean, Google's neural machine translation system: Bridging the gap between human and machine translation, 2016, arXiv abs/1609.08144.

[19] D. Saguil, A. Azim, A layer-partitioning approach for faster execution of neural network-based embedded applications in edge networks, IEEE Access 8 (2020) 59456–59469, http://dx.doi.org/10.1109/ACCESS.2020.2981411.

[20] F. Martins Campos de Oliveira, E. Borin, Partitioning convolutional neural networks to maximize the inference rate on constrained IoT devices, Future Internet 11 (10) (2019).

[21] F. Martins Campos de Oliveira, E. Borin, Partitioning convolutional neural networks for inference on constrained Internet-of-Things devices, 2018, http://dx.doi.org/10.1109/CAHPC.2018.8645927.

[22] A.H. Li, A. Sethy, Semi-supervised learning for text classification by layer partitioning, in: ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, 2020, pp. 6164–6168.

[23] J. Ko, T. Na, M. Amir, S. Mukhopadhyay, Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained Internet-of-Things platforms, in: 2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS, 2018, pp. 1–6.

[24] L. Zhou, H. Wen, R. Teodorescu, D.H. Du, Distributing deep neural networks with containerized partitions at the edge, in: 2nd USENIX Workshop on Hot Topics in Edge Computing, HotEdge 19, USENIX Association, Renton, WA, 2019, URL https://www.usenix.org/conference/hotedge19/presentation/zhou.

[25] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, Y. Chen, HyPar: Towards hybrid parallelism for deep learning accelerator array, in: 2019 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2019, pp. 56–68, http://dx.doi.org/10.1109/HPCA.2019.00027.

[26] J. Zhan, J. Zhang, Pipe-torch: Pipeline-based distributed deep learning in a GPU cluster with heterogeneous networking, in: 2019 Seventh International Conference on Advanced Cloud and Big Data, CBD, 2019, pp. 55–60.

[27] B. Yang, J. Zhang, J. Li, C. Ré, C.R. Aberger, C.D. Sa, PipeMare: Asynchronous pipeline parallel DNN training, 2019, arXiv abs/1910.05124.

[28] J. Geng, D. Li, S. Wang, ElasticPipe: An efficient and dynamic model-parallel solution to DNN training, in: Proceedings of the 10th Workshop on Scientific Cloud Computing, 2019.

[29] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D.R. Burdick, S. Vaithyanathan, Hybrid parallelization strategies for large-scale machine learning in systemml, Proc. VLDB Endow. 7 (7) (2014) 553–564.

[30] J. Ono, M. Utiyama, E. Sumita, Hybrid data-model parallel training for sequence-to-sequence recurrent neural network machine translation, in: PSLT@MTSummit, 2019.

[31] S.B. Akintoye, L. Han, X. Zhang, H. Chen, D. Zhang, A hybrid parallelization approach for distributed and scalable deep learning, 2021, arXiv abs/2104.05035.

[32] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q.V. Le, Z. Chen, GPipe: Efficient training of giant neural networks using pipeline parallelism, 2019, arXiv abs/1811.06965.

[33] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N.R. Devanur, G.R. Ganger, P.B. Gibbons, M. Zaharia, PipeDream: Generalized pipeline parallelism for DNN training, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–15.

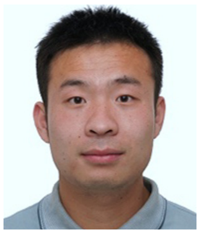
[34] E. Belilovsky, M. Eickenberg, E. Oyallon, Decoupled greedy learning of CNNs, in: Proceedings of the 37th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 119, PMLR, 2020, pp. 736–745.

[35] Z. Huo, B. Gu, Q. Yang, H. Huang, Decoupled parallel backpropagation with convergence guarantee, in: ICML, 2018.
[36] A. Xu, Z. Huo, H. Huang, On the acceleration of deep learning model parallelism with staleness, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 2085–2094, http://dx.doi.org/10.1109/CVPR42600.2020.00216.
[37] W. Dai, Y. Zhou, N. Dong, H. Zhang, E.P. Xing, Toward understanding the impact of staleness in distributed machine learning, in: 7th International Conference on Learning Representations, ICLR, 2019, New Orleans, la, USA, May 6–9, 2019, OpenReview.net, 2019, URL https://openreview.net/forum?id=BylQV305YQ.
[38] R.M. Gower, N. Loizou, X. Qian, A. Sailanbayev, E. Shulgin, P. Richtárik, SGD: General analysis and improved rates, 2019, arXiv abs/1901.09401.
[39] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2015, arXiv abs/1412.6980.
[40] X. Zhang, L. Han, W. Zhu, L. Sun, D. Zhang, An explainable 3D residual self-attention deep neural network FOR joint atrophy localization and Alzheimer's disease diagnosis using structural MRI, IEEE J. Biomed. Health Inform. PP (2021).
[41] D. Sarwinda, R.H. Paradisa, A. Bustamam, P. Anggia, Deep learning in image classification using residual network (ResNet) variants for detection of colorectal cancer, Procedia Comput. Sci. 179 (2021) 423–431, http://dx.doi.org/10.1016/j.procs.2021.01.025, 5th International Conference on Computer Science and Computational Intelligence 2020, URL https://www.sciencedirect.com/science/article/pii/S1877050921000284.
[42] J. Tian, D. Yung, Y.-C. Hsu, Z. Kira, A geometric perspective towards neural calibration via sensitivity decomposition, 2021, arXiv abs/2110.14577.
[43] C. Giuseppe, A ResNet-50-based convolutional neural network model for language ID identification from speech recordings, in: Proceedings of the Third Workshop on Computational Typology and Multilingual NLP, 2021.
[44] S. Sharma, K. Guleria, S. Tiwari, S. Kumar, A deep learning based convolutional neural network model with VGG16 feature extractor for the detection of alzheimer disease using MRI scans, Meas.: Sens. 24 (2022) 100506, http://dx.doi.org/10.1016/j.measen.2022.100506, URL https://www.sciencedirect.com/science/article/pii/S2665917422001404.
[45] F.O. Giuste, J.C. Vizcarra, CIFAR-10 image classification using feature ensembles, 2020, arXiv abs/2002.03846.
[46] PyTorch, Pytorch-deep learning framework that puts python first, 2020, URL http://pytorch.org/.
[47] Nvidia, NCCL, 2021, URL https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html.

Samson B. Akintoye received the Ph.D. degree in Computer Science from University of the Western Cape, South Africa, 2019. He is currently working as a research associate in the Department of Computing and Mathematics, Manchester Metropolitan University, United Kingdom. His current research interests include parallel and distributed computing, machine learning, data science, deep learning, and cloud computing.

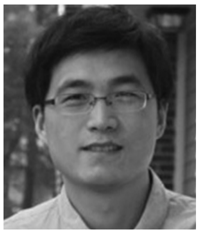
Liangxiu Han received the Ph.D. degree in computer science from Fudan University, Shanghai, China, in 2002. She is currently a professor of computer science with Department of Computing and Mathematics, Manchester Metropolitan University. Her research areas mainly lie in the development of novel big data analytics and development of novel intelligent architectures that facilitates big data analytics (e.g., parallel and distributed computing, Cloud/Service-oriented computing/data intensive computing) as well as applications in different domains using various large datasets (biomedical images, environmental sensor, network traffic data, web documents, etc.) She is currently a Principal Investigator or Co-PI on a number of research projects in the research areas mentioned above.

Huw Lloyd received the B.Sc. degree in Physics from Imperial College, London, UK and the Ph.D. degree in Astrophysics from the University of Manchester, UK. He is currently a Senior Lecturer with the Department of Computing and Mathematics, Manchester Metropolitan University. His research interests cover a range of theoretical and applied topics in machine learning, evolutionary computation, combinatorial and continuous optimisation.


Xin Zhang is associate researcher in Manchester Metropolitan University (MMU), he received the B.S degree from The PLA Academy of Communication and Commanding, China, in 2009 and Ph.D. degree in Cartography and Geographic Information System from Beijing Normal University (BNU), China, in 2014. His current research interests include remote sensing image processing and deep learning.

Darren Dancey is the Head of the Department for Computing and Mathematics. He has a research background in Artificial Intelligence with a Ph.D. in Artificial Neural Networks. His teaching interests centre on Software Development. He has recently taught courses in Data Structures and Algorithms, Comparative Programming languages and applied courses such as Website Development and Mobile Application Development. In recent years, he has concentrated on creating collaborations and knowledge exchange between universities and industry with a focus on the SME sector. He has led several large projects funded by Innovate UK, the Digital R\&D Fund for the Arts, and the European Research Council. He is on the organising committee for the Manchester Raspberry Pi Jam and sits on the BCS Manchester branch committee.

Haoming Chen is studying for a master's degree in Computer Science and Artificial Intelligence in University of Sheffield. His current research interests include machine learning and Artificial Intelligence.


Daoqiang Zhang received the B.Sc. and Ph.D. degrees in computer science from Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1999 and 2004, respectively. He is currently a Professor in the Department of Computer Science and Engineering, Nanjing University of Aeronautics and Astronautics. His current research interests include machine learning, pattern recognition, and biomedical image analysis. In these areas, he has authored or coauthored more than 100 technical papers in the refereed international journals and conference proceedings.