

Parallel and Memory-Efficient Distributed Edge Learning in B5G IoT Networks

Jianxin Zhao, Pierre Vandenhove, Peng Xu, Hao Tao, Liang Wang, Chi Harold Liu, *Senior Member, IEEE*, and Jon Crowcroft, *Fellow, IEEE*

Abstract—Nowadays we are witnessing rapid development of the Internet of Things (IoT), machine learning, and cellular network technologies. They are key components to promote wireless networks beyond 5G (B5G). The plenty of data generated from numerous IoT devices, such as smart sensors and mobile devices, can be utilised to train intelligent models. But it still remains a challenge to efficiently utilise IoT networks and edge in B5G to conduct model training. In this paper, we propose a parallel training method which uses operators as scheduling units during training task assignment. Besides, we discuss a pebble-game-based memory-efficient optimisation in training. Experiments based on various real world network architectures show the flexibility of our proposed method and good performance compared with state of the art.

Index Terms—edge learning, Beyond 5G, memory efficient, backpropagation

I. INTRODUCTION

RECENTLY we have seen rapid development of Internet of Things (IoT) and cellular network technologies such as 5G. These systems incorporate numerous IoT devices, such as smart sensors on cars, mobile phones, cameras, wearables, etc. These IoT devices both generate massive amounts of data and can serve as computation and sensing units in distributed computing systems. Beyond 5G (B5G) networks aims to integrate these heterogeneous IoT devices while meeting a plethora of service requirements [1] [2].

As shown in Fig. 1, distributed edge learning is a paradigm that can benefit from B5G. Here the edge servers utilise IoT networks to jointly train intelligent models, using their local data and computation resources [3]. By moving training jobs from remote cloud servers to edges, edge learning can greatly reduce computation latency and communication traffic. Meanwhile, the cloud servers can utilise the learning models on edge to provide intelligent services while providing overall control. As a result, this paradigm has achieved a wide range of applications in various fields, including unmanned aerial vehicle [4], virtual reality [5], communication [6] [7], transportation [8], and signal processing [9].

J. Zhao, X. Peng and C. H. Liu are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, CN. Email: {jianxin.zhao, chihliu}@bit.edu.cn, xupeng_mii@163.com.

P. Vandenhove is with F.R.S.-FNRS, UMONS - Université de Mons, Belgium, and Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, France. This work was partly done when he was affiliated with the OCaml Labs, University of Cambridge. Email: pierre.vandenhove@umons.ac.be.

H. Tao is with the China Ship Development and Design Center, Wuhan, CN. E-mail: chst722@163.com.

J. Crowcroft and L. Wang are with the University of Cambridge, Cambridge, UK. Email: {jon.crowcroft, liang.wang}@cl.cam.ac.uk

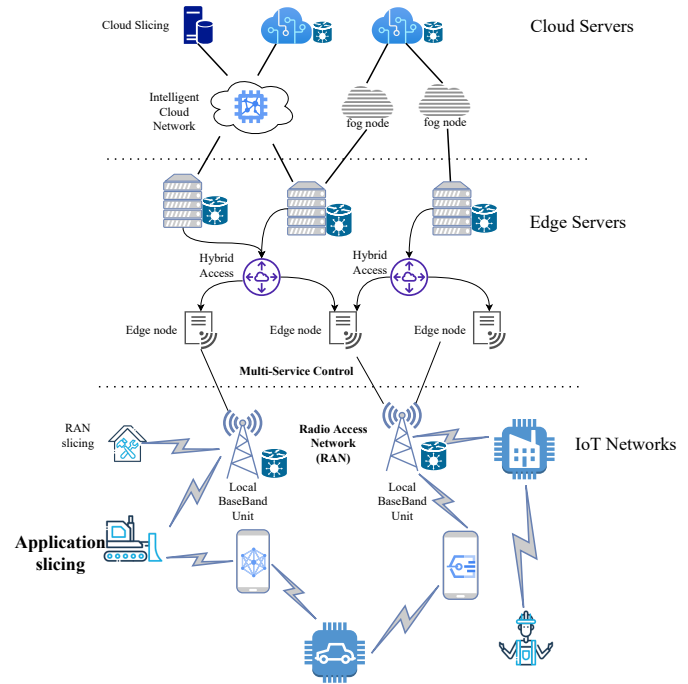


Fig. 1. Architecture of Beyond 5G IoT networks.

However, edge learning also faces challenges from several aspects, most notably the limited computation and memory resources [10]. Despite the potential massive data and computation resources available in an edge learning system, the resource on each single device tends to be quite limited. In contrast, training a machine learning model of even moderate size requires a large amount of resources. To overcome these challenges, existing work follows two general approaches. The first is to reduce the requirement for sources, such as model compression, trading off model accuracy, etc. [11]. The second is to utilise the benefit of a number of devices and distribute workload to multiple edge devices by using paradigms such as federated learning [12] [13].

Specifically, one main research field along this direction is to investigate parallel backpropagation. The backpropagation algorithm implies a strong sequential dependency among neighbouring layers, and thus hinders parallel execution. With the fast development of parallel processing support on hardware, utilising such ability is key to increasing the performance of model training in edge learning [14]. There are several research efforts to solve this problem. Some try to reduce the serial dependency by introducing an approximate algorithm,

where stale values or estimated values are applied on certain layers [15] [16] [17]. Some explore the potential of parallelisation of the computation graph in backpropagation [18]. There are also some works that utilise new structures to improve the backpropagation algorithm itself [19]. However, these works have not yet fully investigated the potential of operator level parallelism.

Toward this end, we propose OLPB, an Operator-Level Parallel scan Backpropagation method in the training of models in edge learning. To utilise IoT nodes to parallelise model training, OLPB combines both aforementioned approaches: partitioning computation on multiple parallel units, and reducing its memory usage and computation complexity on each node. Specifically, we make the following contributions in this paper:

- we propose a parallel training method which partitions workload at operator level, and utilises IoT nodes to perform edge learning;
- we implement an algorithmic differentiation framework to support the proposed backpropagation algorithm; specifically, we formulate the memory reduction problem during the execution of backpropagation as a pebble game, and propose a memory-efficient solution;
- we present a thorough description of the implementation of the proposed method in a scientific computing library;
- we use various real world deep neural network architectures to evaluate the effectiveness of the proposed algorithm.

The rest of this paper is structured as follows. Sec. II provides a literature review of related works about edge learning and parallel training. In Sec. III we discuss an example that motivates the core idea of this paper, and then we explain it in detail in Sec. IV. In Sec. V we focus on techniques we have employed to reduce memory usage and computation complexity, followed by details about its implementation in a scientific library in Sec. VI. In Sec. VII we evaluated the proposed method on a series of neural networks to prove its efficiency compared with state of the art. Finally in Sec. VIII we conclude the whole paper.

II. BACKGROUND

A. Edge Learning and Distributed Training

Federated Learning is a paradigm that allows machine learning tasks to take place without requiring data to be centralised, and therefore is suitable to be applied in edge learning [20]. [21] proposes a privacy-preserving asynchronous federated learning mechanism for edge learning, which allows multiple edge nodes to achieve more efficient federated learning without sharing their private data. [22] introduces a hierarchical federated edge learning framework, where model aggregation is partially migrated to edge servers from the cloud. By relieving core network transmission overhead, it enables great potentials in low-latency and energy-efficient edge learning.

Many works focus on addressing the challenges of energy cost and computation resource limit. [23] proposes to power devices using wireless power transfer. This work aims at the derivation of the tradeoff between the model convergence and

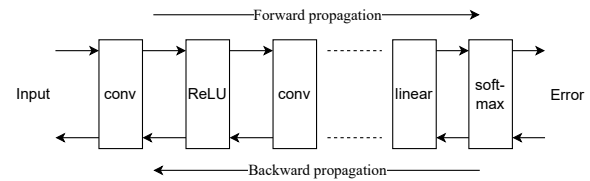


Fig. 2. The forward and backward propagation phases in training a neural network model.

the settings of power sources in different scenarios. To reduce the cost of consuming intelligent edge services, [24] proposes a meta-learning based scheme that adaptively chooses the appropriate machine learning algorithms and resource consumption in different scenarios, based on the meta-features extracted from local data.

In distributed learning, multiple edge devices collaboratively train a model. Stochastic Gradient Descent is one of the most widely used training methods. It iteratively optimises a given objective loss function until convergence. In each iteration, a descent gradient is typically calculated using a batch of training data; the model parameters are updated by moving following the direction of the gradient at a certain point in the vector space [25]. It is widely applied in fields such as edge learning, wireless communication systems, and IoT, etc. [26] [27].

Parameter Server [28] (PS) is a frequently used distributed training architecture. The server keeps the model parameters; the workers *pull* the parameters, compute the gradients, and *push* them back to the server for aggregation. Compared to PS, The Ring All-Reduce architecture organises workers as a ring structure to utilise the bandwidth effectively. The Horovod [29] framework provides a high performance implementation of the All-Reduce. Besides these two, the decentralised architecture has drawn more attention. Though the Peer-to-peer approach can effectively solve problems such as communication bottleneck, unfairness caused by information concentration, etc., a lot of challenges remain to be addressed to train a model with good performance in a decentralised system. For example, [30] investigates the impact of the density of the network topology on the convergence speed.

B. Parallel Backpropagation

Automatic differentiation (AD) is a key technique that enables backpropagation, since the latter is in essence calculating the derivative of the whole neural network as a large function. AD is a computer-friendly technique for performing differentiation that is both efficient and accurate, and is core to many scientific applications [31]. [32] proposes a generalised AD algorithm that is inherently parallel-friendly.

The training of a neural network consists of multiple rounds of optimisation, each mainly containing a forward propagation phase and a backward propagation phase, as shown in Fig. 2. A typical deep neural network consists of multiple layers, such as convolution layer, ReLU activation layer, fully-connected layer, etc. The backpropagation algorithm [33] repeatedly adjusts the weights of each layer so as to minimise the loss value.

Traditional approaches to accelerate backpropagation includes utilising hardwares to improve the performance of calculation, or using the data parallel approach. To exploit the maximum performance of GPU, [34] focuses on improving the performance of each operation on GPU, relying on the CUDA Basic Linear Algebra Subroutines (cuBLAS) function to perform fast matrix and vector operations on CUDA kernel. [35] presents a parallelised backpropagation neural network based on MapReduce computing model and the data parallel paradigm.

Some work seek to revise or change the backpropagation algorithms altogether. The authors of [19] propose to use the feedback alignment method to break the sequential requirement. It shows that the weights used for error backpropagation do not need to be symmetric with the weights used for forward propagation in activation layers. Based on this principle, this work trains hidden layers more independently from the rest of the network. [36] further extends the application of feedback alignment to several new fields, including recommender systems, geometric learning, neural view synthesis, natural language processing, etc.

There are also works that trade off some accuracy in return for breaking the sequential reliability. The authors of [15] propose an approximate backpropagation scheme that works specifically for models that accept redundant input streams as data, such as video. It proposes to overwrite network activations whenever new frames become available, thus breaking the sequential relationship between forward and backward computation. The authors of [16] observe that the Softmax and ReLU outcomes in forward pass for the same labels tends to be quite similar, and thus propose a CNN parallel training method which performs forward and backward propagations simultaneously. [17] decouples the reliance among different phases of backpropagation using gradients from previous training rounds as input, and provides convergence guarantee for the proposed parallel backpropagation algorithm in deep learning optimisation.

Another research direction focuses on utilising the properties of graphs in investing the possibility of parallelism. [37] abstracts parallel training as computation graph transformation problems, so that parallel computation can be achieved by graph manipulation techniques such as duplication, splitting, augmentation, etc. Various parts of the graph can then be assigned to different accelerators. [18] formulates the backpropagation in neural network training process as a generic scan operation, which is a primitive that performs an in-order aggregation on a sequence of values and returns the partial result at each step. This operation can then be efficiently executed on parallel systems.

However, existing works have not fully investigated the potential of parallel backpropagation at the computation operation granularity. They mostly focus on using the neural network layer as a parallel computing unit.

III. MOTIVATION

A neural network can be formally expressed as a series of compounding functions:

$$g(x) = f_{\theta_n}^{(n)}(f_{\theta_{n-1}}^{(n-1)}(\dots f_{\theta_1}^{(1)}(x)))$$

Here $g(x)$ is a neural network that accepts x as input; x can be a 4-dimensional array that represents a batch of images. The network consists of n layers of neuron, and function $f_{\theta_i}^{(i)}$ indicates the i -th layer, e.g. a linear layer, a convolution layer, or a ReLU activation layer. Vector θ_i corresponds to the parameters contained in the i -th layer. For example, a linear layer contains two matrix parameters: the weight (w) and bias (b), and performs the transformation $wx + b$ on input x .

In a training process, the input is propagated through this function (forward propagation) to calculate $yt = g(x)$, which is then compared with a known “true” y value in a certain loss function to calculate an error value. Next, the error value is backward-propagated throughout the network to find the gradients for each parameter. According to the gradient descent method, the parameters are then optimised following the direction of the gradient values. The training process consists of several such iterations until convergence. Specifically, the gradient of parameter θ_i is denoted $\frac{\partial g}{\partial \theta_i}$. According to the chain rule,

$$\begin{aligned} \frac{\partial g}{\partial \theta_i} &= \frac{\partial g}{\partial f_i} \frac{\partial f_i}{\partial \theta_i} \\ &= \frac{\partial g}{\partial f_{n-1}} \frac{\partial f_{n-1}}{\partial f_{n-2}} \dots \frac{\partial f_{i+1}}{\partial f_i} \frac{\partial f_i}{\partial \theta_i} \end{aligned} \quad (1)$$

The partial derivative $\frac{\partial f_i}{\partial \theta_i}$ is a local computation that solely depends on function f_i . However, the sequence before this item in Eq. (1) indicates a strong sequential dependency. To compute $\frac{\partial g}{\partial \theta_i}$, one must first compute $\frac{\partial g}{\partial \theta_{i+1}}$.

To address this issue, [18] proposes to model the previous training process as a series of scan operations:

$$\left[\frac{\partial g}{\partial f_{n-1}}, \frac{\partial g}{\partial f_{n-2}}, \dots, \frac{\partial g}{\partial f_1} \right] = \left[\frac{\partial f_n}{\partial f_{n-1}}, \frac{\partial f_n}{\partial f_{n-1}} \frac{\partial f_{n-1}}{\partial f_{n-2}}, \dots, \right. \\ \left. \frac{\partial f_n}{\partial f_{n-1}} \frac{\partial f_{n-1}}{\partial f_{n-2}} \dots \frac{\partial f_2}{\partial f_1} \right] \quad (2)$$

Here one element in the list is prefix of the later one. Based on this property in this sequence, the basic idea of a parallel scan algorithm is that the computation of one item can be utilised to compute some other items. As a result, the parameters θ_i , $i = 1, 2, \dots, n$ can be computed in parallel given sufficient computation threads.

In this work, the parallel algorithm utilises backpropagation of different layers as a computation unit. However, we observe that in a neural network, each layer consists of multiple operations, and thus could potentially enable parallel computing at finer granularity.

Fig. 3 shows a simple example. A convolution neural layer consists of two operations: convolution and add, and two parameters: w and b . Here w is the kernel parameter in convolution. In the forward propagation phase, the convolution operation accepts two inputs; the first is output from previous layers, such as a ReLU activation layer, and the second is its kernel parameter. In the backward propagation phase, the

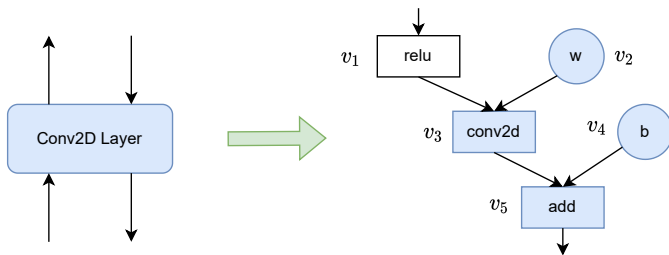


Fig. 3. The operations and parameters included in a two-dimensional convolution layer in a neural network.

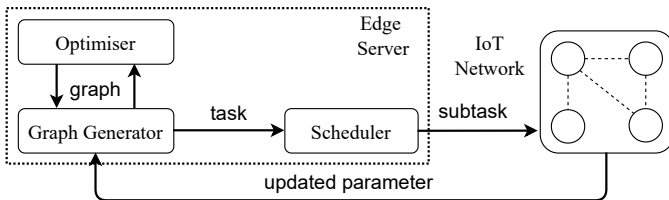


Fig. 4. System overview of the proposed operator-level parallel backpropagation framework.

error is propagated in a bottom-up manner. We have denoted each node as v_i , $i \in [1, 5]$, and the propagated error at v_i is denoted as $\tilde{v}_i = \frac{\partial v_i}{\partial v_5}$. Suppose the initial error at v_5 is 1, then the backpropagation process in this subgraph is as follows:

- $\tilde{v}_5 = 1$
- $\tilde{v}_4 = \tilde{v}_5 \frac{\partial(v_3+v_4)}{\partial v_4}$
- $\tilde{v}_3 = \tilde{v}_5 \frac{\partial(v_3+v_4)}{\partial v_3}$
- $\tilde{v}_2 = \tilde{v}_3 \frac{\partial \text{conv}(v_1, v_2)}{\partial v_2}$
- $\tilde{v}_1 = \tilde{v}_3 \frac{\partial \text{conv}(v_1, v_2)}{\partial v_1}$

The backpropagation at the operator level still follows the same approach as that at the layer level. Besides, there is no distinction between the gradients of operators and parameters. The only difference is that the propagation path is now a tree structure instead of a linear one. It provides ample optimisation space for parallel computing. For example, once the calculation on v_5 finishes, v_3 and v_4 can start theirs in parallel. Motivated by this observation, we propose our parallel backpropagation method in the next section.

IV. OPERATOR-LEVEL PARALLEL BACKPROPAGATION

In this section we present Operator-Level Parallel Backpropagation (OLPB), a framework that flexibly executes training of machine learning models utilising parallel execution in IoT networks. Fig. 4 provides a system overview of this framework. It mainly contains four components: graph generator, optimiser, scheduler, and IoT computing nodes. The first three parts are included in the edge server, and the computing node constitutes an IoT network. In general, this framework generates the computation graph used in training, optimises it, splits it onto multiple nodes, and then performs parallel computation. Next we will introduce these components in detail.

Graph generator. The main training process in the edge server follows the normal steps. First it chooses to use certain network architecture, and sets hyper-parameters such as batch

size, epochs, learning rate, etc. Then the training contains several iterations. In each iteration, the server (a) performs a forward propagation on the neural network to get an inference output, and calculate its “distance”, or error value with regards to a ground truth value using a certain loss function; (b) backpropagates the error throughout the neural network reversely, calculating gradients of each parameter regarding the error; (c) updates neural network parameters according to a learning method (e.g. SGD, Adagrad, etc.) and the gradients.

In a second step, the edge server sends the graph generated for backpropagation to the *Task Scheduler* for parallel execution. Optionally, it can choose to first pre-process the graph in the *Optimiser* module to optimise its graph architecture and memory efficiency. We will discuss this component in detail in the next section.

Task Scheduler. The aim of the scheduler is to divide a computation graph into multiple fragments, each as a subtask, and assign them to workers (IoT nodes in our scenario) in a proper way. This module includes two aspects: first, the unit of scheduling, and then how to partition these units to a pool of workers.

When choosing the unit of scheduling, it is natural to consider the neural network layer. It is intuitive to understand, and many deep neural networks stack the neurons layer by layer linearly. Many deep learning frameworks such as PyTorch also provide high level API to interact directly with the forward and backward propagation. As shown in Sec. III, the backpropagation process as the computation sequence can be formalised as below:

$$\left[\frac{\partial g}{\partial f_{n-1}}, \frac{\partial g}{\partial f_{n-2}}, \dots, \frac{\partial g}{\partial f_1} \right]$$

where f_i represents a layer in a neural network.

In this work we instead use the operator as a schedule unit, which provides finer granularity of control. For example, a `BatchNormalisation` layer may contain about 7 operator nodes, and treating them as different units can benefit from the parallel computing paradigm more flexibly. However, one challenge is that a computation graph with the operator as the basic unit often forms a tree structure rather than being linear. In fact, the aforementioned model faces challenges even at the neural network layer granularity if the architecture is not strictly linear, e.g. in Inception DNN architectures [38]. To that end, we introduce the primitive ι :

Definition 1: Let A and B be sets that contains nodes in a computation graph G , then $A \iota B$ is defined as $\{a'_i \frac{\partial b_i}{\partial a_i} | \forall a_i \in A, b_j \in B, \text{ and } b_j \text{ is a parent of } a_i\}$. The primitive ι is binary, associative, and non-commutative. Here a'_i is the derivative of a_i .

Using ι , we can now describe the scheduling method in OLPB. First, the scheduler tags the nodes in a graph by its level in the tree. For example, a leaf node is tagged as level 1, its parents are at level 2, and their parents are at level 3, etc. If a node is parent to multiple nodes of different levels, it should be tagged according to whichever is at a higher level. Thus, the calculation of operators in level n relies on those in level $n - 1$.

After the tagging process, the whole network consists of different levels, which connect with each other linearly, stacking similarly as the neural network layers. Denoting each layer as G_i , the scan operation can be formalised as:

$$[G_1, G_1 \iota G_2, \dots, G_1 \iota \dots \iota G_n] \quad (3)$$

One notable benefit of this approach of organising operators is the flexibility it provides. Putting nodes of the same layer as the same group is one way to perform tagging, but it is not necessarily the only way. Let's consider how "fragmented" a computation graph division should be. On one hand, if divided into too few fragments, we cannot fully benefit from parallelism. On the other hand, if we dissect the network into too many small pieces, such as each node assigned to a worker node, the communication costs among IoT nodes could rise greatly. Besides, it is not time-efficient. The reason is that, one notable feature of the backpropagation in a neural network is that the execution time of various operators are highly biased. Operations such as convolution, ReLU, multiplication, etc. make up a large proportion of the whole running time, with a long tail that contains the other light-weight operations such as summation. Therefore, e.g. if one convolution and multiple add or multiplication operations are executed in parallel, it is quite likely that the latter ones complete their subtasks early and have to wait for the convolution subtask to finish. Our proposed OLPB thus enables exploring this tradeoff about task scheduling.

Parallel Execution in IoT Network. The grouped computation graph nodes, or subtasks, are assigned to the available IoT nodes. The scheduler first queries the IoT nodes about their availability, giving conditions such as network connection, computation resource, storage, etc. Each subtask is assigned to an IoT device that fits its requirement.

If the number of computing nodes is more than that of the subtasks, these subtasks are executed in batches in the IoT network. In this batch, there are n parallel computing threads, and the target is to compute Eq. (3). Here each G_i is the set of nodes contained in subtask that is assigned to IoT node i .

Computing the n elements in Eq. (3) can be modelled as a general form of scan process. The scan of an array of input $[x_1, \dots, x_n]$, given an operator ι , computes the sequence: $x_1, x_1 \iota x_2, \dots, x_1 \iota x_2 \dots \iota x_n$. It is a common parallel algorithm building block [39], and there exists several parallel scan algorithms. Here we choose the one proposed by Hillis and Steele [40]. The intuition is to use inter-thread communication and utilise the existing result of one thread as an intermediate result for other threads. Its mechanism is illustrated in Fig. 5. It shows performing parallel scan on a 8-element list in three iterations. In the first iteration, every two adjacent threads aggregate their results. At the next iteration, the two threads with a distance of 2, such as the first and the third, aggregate the result. And aggregation at the third iteration happens between threads with a distance of 4, etc.

To sum up what we have introduced, Alg. 1 and Alg. 2 show the proposed OLPB algorithm. Here $\log(N)$ corresponds to the iteration number required to finish the client side algorithm, and l indicates the aforementioned *level* of operators.

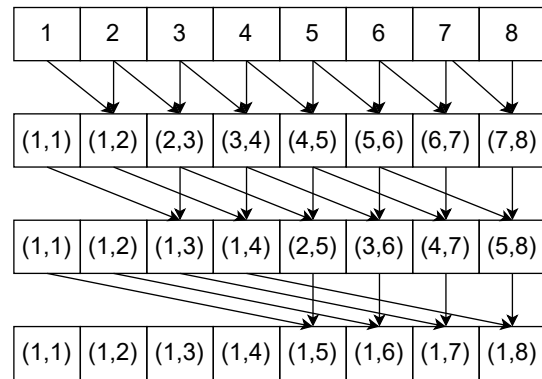


Fig. 5. An example of Steele parallel scan algorithm. It contains eight threads, and computes for three iterations.

Algorithm 1 Pseudo-code of the OLPB executed on server side.

- 1: **Input:** neural network \mathcal{N} , input data d , loss function f , level parameter l
- 2: **Output:** subtasks $[g_1, g_2, \dots, g_M]$
- 3: **procedure** SERVERPROCEDURE
- 4: graph $G = f(\mathcal{G}, d)$
- 5: optimise G
- 6: tag operator nodes in G by their levels in the graph
- 7: partition graph nodes in G to subtasks by tag and l
- 8: **end procedure**

Algorithm 2 Pseudo-code of the OLPB executed on IoT client side. Inspired by Hillis and Steele [40].

- 1: **Input:** subtasks $s = [g_1, g_2, \dots, g_N]$
- 2: **Output:** gradients $[\partial g_1, \partial g_2, \dots, \partial g_N]$
- 3: **procedure** CLIENTPROCEDURE
- 4: **for** $h = 1$ to $\log(N)$ **do**
- 5: **for** $k = 1$ to N **do in parallel**
- 6: **if** $k \geq 2^h$ **then**
- 7: $s[k] \leftarrow s[k] \iota s[k - 2^{h-1}]$
- 8: **end if**
- 9: **end for**
- 10: **end for**
- 11: **Return** s
- 12: **end procedure**

To measure the efficiency of this algorithm, we use two metrics. The first is *step complexity* (S), which measures the minimum number of steps to finish the execution. The second is *work complexity* (W) that evaluates the number of total steps executed by all worker nodes. Denoting the number of IoT nodes as b and the number of subtasks generated in the scheduler as n , the complexities of our parallel algorithm is: $S = \frac{n}{b} + \log(b)$, and $W = n \log(n)$.

V. MEMORY-EFFICIENT COMPUTATION GRAPH OPTIMISER

The implementation is on Owl, an OCaml functional programming language based scientific computing library [41]. It allows writing succinct type-safe numerical applications in a concise functional language without sacrificing performance. Owl provides an n -dimensional array (ndarray) module, based

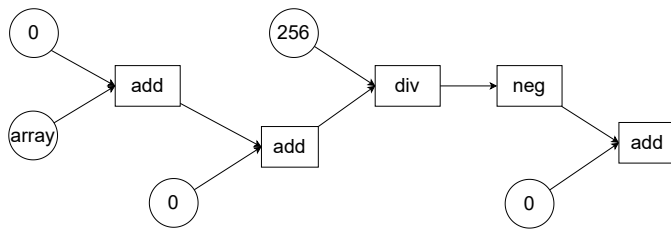


Fig. 6. Optimisation techniques in computation graph: constant folding. All nodes in this figure can be folded into one single node.

on which we implement the algorithmic differentiation module and the proposed OLPB method. The backpropagation calculation is conducted in the form of a computation graph. We have implemented two forms of optimisation on it to reduce the memory consumption: graph architecture optimisation, and a pebble-game-based memory-usage optimisation. We will introduce them in this section.

A. Graph Structure Optimisation

The graph optimiser searches for certain structural patterns in a graph, and then performs various optimisations, such as removing unnecessary computations, fusing computation nodes, etc. We have implemented the commonly used graph optimisation patterns, including constant folding, operations fusing, removing zeros, etc. All the patterns are defined in the Owl computation Optimiser module. The code is organised so that it is straightforward to plug in more patterns to extend the Optimiser's capability.

The code below shows the core function of this module. This function starts from the leaves of a computation graph, and traverses backward to their ancestors recursively, looking for certain patterns to optimise. The patterns which functors can recognise are coded in various `pattern_*` functions which call each other recursively.

```
let rec optimise_term x =
  match get_operator x with
  | Noop -> pattern_003 x
  | Empty_shape -> pattern_000 x
  | Zeros_shape -> pattern_000 x
  ...
  | Add -> pattern_001 x
  | Sub -> pattern_000 x
  | Mul -> pattern_019 x
  ...
  | Scalar_Add -> pattern_010 x
  | Scalar_Sub -> pattern_010 x
  ...
  | Dot (_transa, _transb, _alpha,
        _beta) -> pattern_005 x
  | Fused_Adagrad (_rate, _eps)
    -> pattern_000 x
  | _ -> failwith "Unsupported op"
```

For example, *constant folding* is a common pattern in the computation graph generated by neural networks. Such graphs often involve a lot of constants. As a result, some subgraphs can be pre-calculated. Fig. 6 shows such an example. In

this subgraph, the nodes are either constants or operations on constants. We can thus fold this subgraph into one node before evaluating the whole graph. From the definition of the optimise term function, we can see the `Scalar_Add` operator triggers `pattern_010` function. This function first tries to optimise the parent nodes, then it checks whether both parents are constants. If so, the function evaluates the expression given the current operator, creates a new constant node for the result, and removes the current node and its parents. By doing so, all the expressions which can be evaluated during this phase will be folded into a constant, which can save execution time and memory usage during the graph evaluation phase.

Other optimisation patterns are also included, such as *fusing operations*, which combines multiple operations into one, and *Adding zero* which removes the zero nodes in computation. As a simple evaluation of the effectiveness of the implemented methods, we use a LeNet-like deep neural network that is used for training on the MNIST dataset. It mainly consists of two convolution layers, with 1x1 and 5x5 kernel sizes separately. Each is followed by a ReLU activation layer and a maxpooling layer. After them are two FullyConnected layers, with output sizes 120 and 84. Finally the output size is reduced to 10 by another linear layer and a Softmax activation. The original network has 201 nodes and 239 edges; after applying the graph optimisation, the whole computation graph consists of only 103 nodes and 140 edges.

B. Memory-Efficient Backpropagation

Besides graph structure optimisation, we have also implemented another important module in backpropagation: memory management. Specifically, we need the ability to pre-allocate a memory space to each node, to decrease the overall memory consumption and reduce the garbage collector overhead. The key is to find an allocated memory block in the graph that is no longer required, and assign it to other nodes that are in need. During this process, we maintain a list of nodes that share the same memory.

As an initial strategy to allocate memory to a node, we simply reuse the memory of a direct predecessor with the same output shape as the node when that is possible. For example, if we add two ndarrays of the same shape, and the result can reuse the memory block of one of its input ndarrays. This strategy is straightforward and easy to implement, but a node can only utilise its direct parents' memory, which limits the potential available memory space.

Based on this insight, we have proposed and implemented a memory allocation strategy that allows each node to utilise a wide range of available memory space in the backpropagation. The proposed method models the memory allocation problem as a *pebble game* [42], which was introduced to explain register allocation. The *pebble game* is played on a directed acyclic graph. Each node can store at most one pebble. The game begins with no pebble on any node. At each step, the player can do one of the following moves:

- 1) if a vertex v has no predecessor, the player can place a pebble on v .

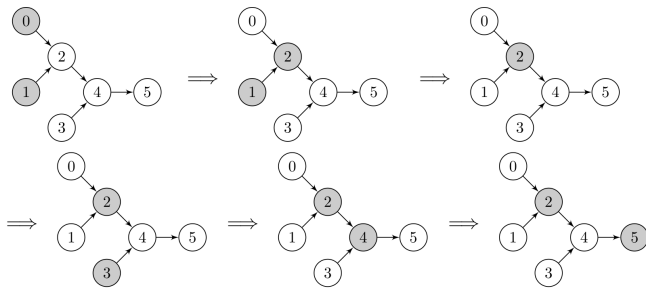


Fig. 7. Modelling computation graph memory optimisation problem as a pebble game

- 2) if all predecessors of a vertex v are pebbled, the player can place a pebble on v or *slide* a pebble from one of its predecessors to v .
- 3) the player can remove any pebble from a vertex (and reuse that pebble later).

The goal of the game is to place a pebble at least once on some fixed output vertices of the graph. Fig. 7 shows an example of an optimal pebbling strategy using the previous computation graph (grey nodes are pebbled), using moves 1) \rightarrow 2) \rightarrow 3) \rightarrow 1) \rightarrow 2) \rightarrow 2). We assume that the goal is to pebble node 5.

This game relates to the memory allocation of the computation graph if we see pebbles as memory blocks used to store the output value of a node. We assume that the values of the inputs are known (move 1). We can only compute the value of a vertex if all its predecessors are simultaneously stored in memory (move 2). The *sliding* move means that the memory of a node can be overwritten by its successor during its computation (*inplace reuse*). We can always reuse a memory block from any other node (move 3). Given a graph, the idea is thus to find a strategy to pebble it using a minimum number of pebbles (in other words, using as little memory as possible).

We also want to avoid pebbling any node twice in order to keep the execution time as low as possible, because that would mean that we compute the same node twice. Given these constraints, finding a strategy using the least amount of pebbles is unfortunately NP-complete [42]. Since computation graphs can have a few thousand nodes, we implement a fast heuristic instead of an exact algorithm.

Now we can apply the pebble game process in our memory allocation process. We propose to share memory between nodes that (1) are not necessarily a parent/child pair, and (2) that do not have the same output size (by allocating a large block of memory once, without necessarily using all of it all the time). To do this efficiently, we first have to fix an evaluation order (in practice, any topological order). Given this order, we can pinpoint the moment when the memory of a node becomes useless by keeping a counter of how many times it has been used. When it has been used by all its children, we can recycle its memory. Then to allocate memory to a node, we simply check which blocks are available and we select the one with the closest size (in order not to waste too much memory). If no block is available, we allocate a new one. This can be

executed in $\mathcal{O}(n * \log(n))$ time, which is negligible compared to the actual cost of evaluating the graph.

Note that some operations cannot overwrite their inputs while they are being computed (the *sliding* move from the pebble game is forbidden) and that some nodes cannot be overwritten for practical purposes, typically constant nodes or neural network weights. When evaluated in the right order, the computation graph needs a much smaller block of memory than the non-optimised version. Specifically, when introducing a new node, if the memory blocks of the parents are reusable, the function checks whether the memory is large enough to accommodate the output of the current operator. If so, the node includes the current operator to the list of nodes sharing the same memory block. Moreover, the memory is reshaped according to the shape of the output.

VI. IMPLEMENTATION

The OLPB is implemented on the Algorithmic Differentiation module in the Owl Scientific Computing Library [43], which is developed by several of this work's authors. Therefore in this section we present the implementation of related modules in this library.

A. Architecture Overview

In this part, we introduce the architecture design of Owl, demonstrate briefly the functionalities of different modules, and aim to explain why we design it this way. We show that Owl benefits greatly from OCaml's module system which not only allows us to write concise generic code with superior performance, but also leads to a very unique design to enable parallel and distributed computing. OCaml's static type checking significantly reduces potential bugs and accelerates the development cycle.

Owl is a rather complex library considering the numerous functions implemented (over 6500 by the end of 2021). We have strived for a modular design to make sure that the system is flexible enough to be extended in future. In the following, we will present its architecture briefly.

Fig. 8 gives a bird's-eye view of Owl's system architecture, i.e. the two subsystems and their modules. The subsystem on the left part is Owl's numerical subsystem. The modules contained in this subsystem fall into four categories:

- core modules contains basic data structures, namely the ndarray; including C-based and OCaml-based implementation, and symbolic representation;
- classic analytics contains basic mathematical and statistical functions, linear algebra, signal processing, ordinary differential equation, etc., and foreign function interface to other libraries (e.g., CBLAS and LAPACK);
- advanced analytics, such as algorithmic differentiation, optimisation, regression, neural network, natural language processing, etc.;
- service composition and deployment to multiple backends such as to browser, container, virtual machine, and other accelerators via computation graph.

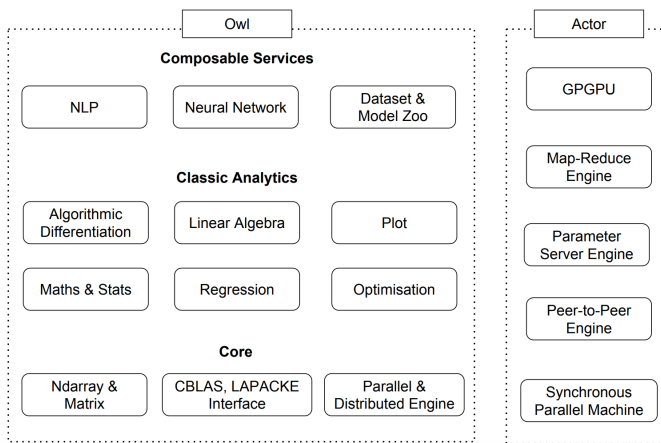


Fig. 8. The whole system can be divided into two subsystems. The subsystem on the left deals with numerical computation, whilst the one on the right handles the related tasks in a distributed and parallel computing context including synchronisation, scheduling, etc.

The subsystem on the right is called Actor subsystem. It extends Owl’s capability to parallel and distributed computing. The addition of the Actor subsystem makes Owl fundamentally different from mainstream numerical libraries such as SciPy and Julia. The core idea is to transform a user application from sequential execution mode into parallel mode (using various computation engines) with minimal efforts. The method we used is to compose two subsystems together with functors to generate the parallel version of the module defined in the numerical subsystem.

B. *N*-dimensional Array

N-dimensional array is the most fundamental data type in a numerical library for building scientific computing applications. Matrices and vectors are special cases of the ndarray. Scalar-based computations are easy to understand but limited in the problems they can solve. Nddarray is the core data structure in many mainstream numerical libraries and software, including NumPy, MATLAB, Julia, and TensorFlow. So it is in Owl, provided in the Nddarray module. It is essential to the whole architecture in Owl. The data types used in Nddarray is built directly on top of OCaml’s native Bigarray data structure used to express large, multi-dimensional numerical arrays and designed to enable efficient sharing of large numerical arrays between OCaml code and C or Fortran numerical libraries.

C. Algorithmic Differentiation in Owl

The AD module plays a pivotal role in the whole Owl library stack. It unifies various fundamental data types in Owl, and is the most important functionality that serves the advanced analytics such as regression, neural networks, etc.

Here we use a simple example to see how the AD module is used in Owl. In this example, we calculate the first order and second order derivatives of the function \tanh .

```
module AD = Algodiff.D
```

```
let f x = AD.Maths.(tanh x);;
let f1 = AD.diff f;;
let f2 = AD.diff f1;;

let eval_flt h x = AD.(pack_flt x
  |> h |> unpack_flt)
```

Let us inspect how various parts in the AD module fit into the example code. First, we cannot directly use the basic data types, such as ndarray and float number. Instead, they need to be first “packed” into a type that the AD module understands. In this example, `pack_flt` is used to wrap a normal float number into an AD type float. After calculation finishes, assuming we still get an AD type float as output, it should be unpacked into a normal float number using the function `unpack_flt`. The **type system** is the most fundamental building block in AD. Second, to construct a computation in the AD system, we need to use **operators**, such as `tanh` used in this example. AD provides a rich set of operators that are generated from the `op_builder` module. After constructing a graph by stacking the operators, the AD engine starts to let the input data “flow”, or “propagate”, twice in this graph, once forward and once backward. The key function in charge of this process is the **reverse** function. It is arguably the most important function that powers the AD system. Based on the aforementioned process, we can calculate differentiation of various sorts. To simplify coding, a series of **high level API** such as `diff` used in this example are constructed. It represents applying the differentiation function on a function that accepts a float number as input and outputs a float number. These high level APIs lead to extremely elegant code. As shown in this example, we can simply apply the differentiation function on the original `tanh` function iteratively to get its first order, second order, and any other higher order derivatives. In the next several sections, we will explain these building blocks in detail and how these different pieces are assembled into a powerful AD module.

VII. EVALUATION

We set up a small-scale evaluation environment to demonstrate the potential performance of our proposed method. We use Docker containers to function as IoT network nodes. The communication delay between two nodes follows that of [44], where it is modelled as an exponential distribution: $t = \text{Exp}(\lambda = 1) * p$. Here we set its rate parameter λ to 1, and the distribution is multiplied by a scale factor p to get the specific delay time.

Baseline We compare our proposed OLPB method with two baseline methods. The first is BPPSA [18], which also performs parallel backpropagation based on generic scan operations. The main difference is that BPPSA only uses each neural network layer as a scheduling unit in parallel computation. This work is state of the art in this topic. The second is decoupled parallel backpropagation (DPB) [17]. It decouples the reliance among different phases of backpropagation using gradients from previous training iterations as input, and is thus an approximate method. We implement our proposed algorithm together with the baseline methods in

TABLE I
INFORMATION ABOUT DEEP NEURAL NETWORK ARCHITECTURES IN THE EVALUATION.

Architecture	#Layers	#Nodes	#Param
CifarNet	17	49	4.8M
SqueezeNet	65	153	4.7M
VGG16	38	97	528M
InceptionV3	313	1097	91M

OCaml language and the Owl scientific computing library. The OCaml version is 4.14.0, and the Owl library version is 1.0.3.

Models To evaluate the proposed method on neural network architectures with sufficient diversity, we use four different types of networks. VGG16 is a classic network that contains 13 convolutional layers, though it also contains a large amount of parameters. The InceptionV3 network is more computationally efficient than VGG, but with much more complex architecture. SqueezeNet is a lightweight architecture that achieves AlexNet-level accuracy on ImageNet dataset with 50x fewer parameters. Besides these networks, we also build a small architecture CifarNet that can be used for training on the CIFAR10 dataset. It contains four convolution layers, two maxpool layers, and two fully-connected layers. These networks vary in layers, number of nodes, and parameters sizes (each element is of single precision). The detailed statistics are shown in Tbl. I.

A. Running Time Comparison

First we compare the time to run backpropagation in a whole computation graph with baseline methods, since it is the most important metric that indicates the parallel execution efficiency. We compare three methods. For our proposed method OLPB, we choose two variants with different level parameters: $l = 2$ and $l = 3$. It indicates how many levels of operator nodes in a computation graph the scheduler groups together as a subtask. The DPB method partitions a whole backpropagation graph into a small number of modules, and they run in parallel. Here we follow the original setting and partition a graph into 4 modules. For the other three methods, we set the number of participating IoT nodes to be 32.

We compare them on four different networks, as shown in Fig. 9. Due to the vast difference in the execution time of these networks, we use the percentage to the slowest method as a metric in the figure. The baseline method DPB runs the slowest, since each of its modules runs on one node, and thus cannot benefit from the extra processing units. OLPB of both levels outperform BPPSA; the time reduction can be as high as 60%. As to the l parameter of OLPB, on the former three networks, level 2 performs better than level 3. For InceptionV3, which is extremely complex, using three levels to build subtasks in the scheduler shows a better performance.

B. Sensitivity Analysis

The number of computing nodes is key to performance of any parallel algorithm. In Fig. 10 we evaluate the impact of this factor on the proposed algorithm. We compare it with the

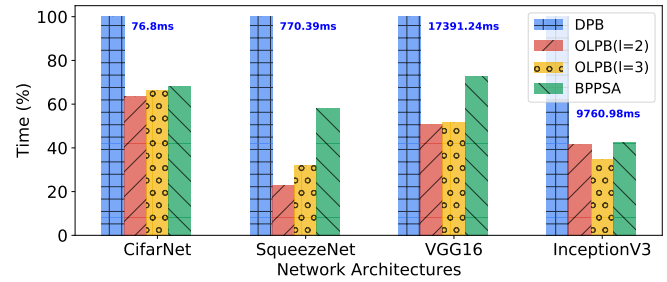


Fig. 9. Execution time of backpropagation with various methods on different network architectures.

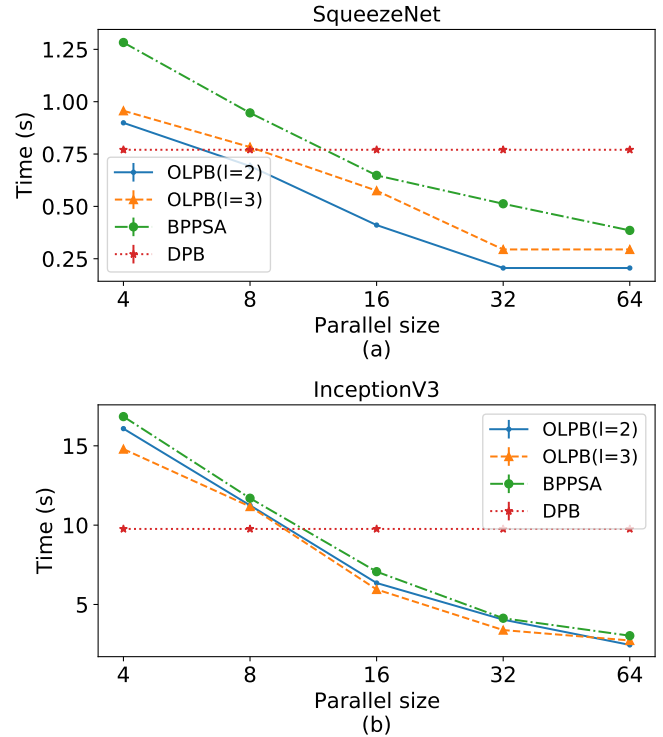


Fig. 10. Impact of parallel size on backpropagation execution time on (a) SqueezeNet and (b) InceptionV3 neural network architectures.

baseline method DPB and BPPSA, and we follow the same configuration as in the previous experiment. Here we vary the number of parallel processing nodes from 4 to 64. The experiment is conducted on both SqueezeNet and InceptionV3. Here DPB is not affected by the number of workers. Actually it gains advantage when the number of nodes is as small as 4, since the other methods are yet to benefit from parallelism. As the number of nodes increases from 4 nodes to 32 nodes, the execution time of OLPB and BPPSA quickly decreases by about 75%. OLPB outperforms BPPSA on both levels, and it can be seen that level 2 is a better choice, being about 50-100ms faster. Changing to InceptionV3 shows a similar trend, though here level 3 OLPB is a better choice due to the complexity of network architecture.

Communication delay is another important factor that affects the performance of parallel backpropagation algorithms. In this experiment we simulate the communication delay with the exponential distribution model: $t = \text{Exp}(\lambda = 1) * p$,

TABLE II
EVALUATION OF THE EFFECT OF CGRAPH MEMORY OPTIMISATION USING DIFFERENT DNN ARCHITECTURES

Architecture	Time (s)	Time w/ optimisation (s) (build + execution) (s)	Memory (MB)	Memory w/ optimisation (MB)
InceptionV3	0.565	0.107 + 0.228 = 0.335	625.76	230.10
ResNet50	0.793	0.140 + 0.609 = 0.749	1309.9	397.07
Mask R-CNN	11.538	0.363 + 8.379 = 8.742	6483.4	870.48

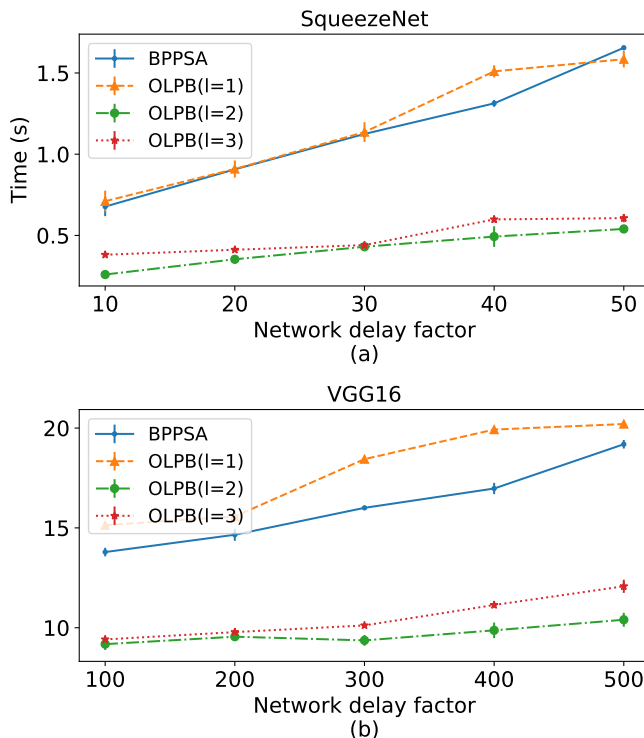


Fig. 11. Impact of communication delay on backpropagation execution time on (a) SqueezeNet and (b) VGG16 neural network architectures.

where p is a delay factor. We keep increasing this delay factor for the calculation on SqueezeNet and VGG16, as shown in Fig. 11. Here we omit the result of the DPB method, since its performance is not obviously affected by communication time. The result shows that both OLPB and BPPSA are affected by the increase of communication time, since that means greater cost of device to device communication delay in the parallel algorithm. But still, the running time with OLPB rises slower than that of BPPSA; the time of OLPB with parameter $l = 2$ increases about 2.07x, from 260ms to 540ms, while that rate of BPPSA is about 2.45x. The result also shows that varying the parameter l parameter in OLPB enables exploring a suitable option for different network architectures.

Finally, we need to examine if changing the parallel scan algorithm and input batch size in training has any impact on the performance of our proposed method. In Fig. 12, we apply two parallel algorithms: the default method proposed by Steele *et al.*, and the method proposed by Blleloch *et al.* Besides, we also vary the batch size of input from 10, to 15 and 20. The evaluation is conducted on SqueezeNet. Here we use 16 nodes for parallel execution, and setting communication delay to 20. The result shows that larger batch size proportionally increases

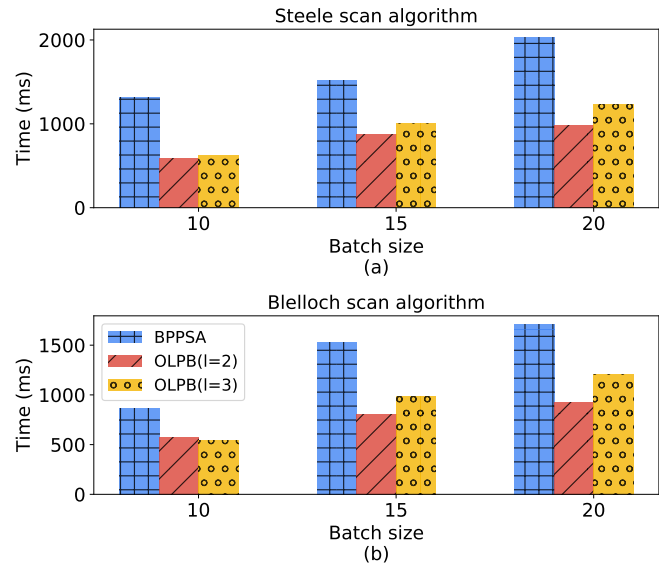


Fig. 12. Impact of batches on the performance of proposed method; execution time comparison on SqueezeNet architecture using (a) Steele parallel algorithm and (b) Blleloch parallel algorithm.

the execution time of BPPSA and OLPB methods by factors from 1.5 to 2 times. Besides, applying a faster scan algorithm can decrease the execution time. But in general the superiority of the proposed OLPB method still holds.

C. Computation and Memory Efficiency

In the last part of this section, we evaluate the effectiveness of our proposed memory optimisation module in the OLPB. We evaluate the inference time of whole networks to compare their performance before and after applying the optimisation phase. These experiments run on a laptop with an Intel i5-6300HQ and 8GB of RAM. In this evaluation, the InceptionV3 and ResNet50 networks are tested with a 299x299 image. We also use Mask R-CNN [45], a network for image instance segmentation. It is tested with a 768x768 image. The final result is calculated as an average over 30 evaluations, without reusing pre-computed nodes when a computation graph is used. The graph building phase includes graph construction, optimisation and memory initialisation. Tbl. II shows statistics illustrating what the computation graph with this new algorithm achieves. Here we consider the time used for building the whole computation graph and execution in the third column. The result clearly shows the performance of the proposed optimisation module. It reduces the memory usage of the Mask R-CNN network from about 6.5GB down to only 870MB, and the execution time also decreases by 25%.

VIII. CONCLUSION

Currently, wireless networks beyond 5G attracts interest from both academia and industry, but it still remains a challenge to efficiently utilise IoT networks and edge in B5G to conduct model training. In this paper, we have proposed a parallel training method which uses operators as scheduling units during training task assignments. And we have also introduced a pebble-game-based memory-efficient optimisation in training. We have conducted a series of experiments based on various real world network architectures, and compared our proposed method OLPB with the state of the art algorithms. The experiment results show the flexibility and good performance of our proposed method.

ACKNOWLEDGMENTS

This work was supported by the National Defence Basic Scientific Research JCKY 2020XXB028.

REFERENCES

[1] J. Zhang, E. Björnson, M. Matthaiou, D. W. K. Ng, H. Yang, and D. J. Love, "Prospective multiple antenna technologies for beyond 5G," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 8, pp. 1637–1660, 2020.

[2] Z. Wang, Z. Liu, Y. Shen, A. Conti, and M. Z. Win, "Location awareness in beyond 5G networks via reconfigurable intelligent surfaces," *IEEE Journal on Selected Areas in Communications*, 2022.

[3] J. Zhang, Z. Qu, C. Chen, H. Wang, Y. Zhan, B. Ye, and S. Guo, "Edge learning: The enabling technology for distributed big data analytics in the edge," *ACM Comput. Surv.*, vol. 54, no. 7, jul 2021. [Online]. Available: <https://doi.org/10.1145/3464419>

[4] S. Tang, W. Zhou, L. Chen, L. Lai, J. Xia, and L. Fan, "Battery-constrained federated edge learning in UAV-enabled IoT for B5G/6G networks," *Physical Communication*, vol. 47, p. 101381, 2021.

[5] H. Xiao, C. Xu, Z. Feng, R. Ding, S. Yang, L. Zhong, J. Liang, and G.-M. Muntean, "A transcoding-enabled 360° VR video caching and delivery framework for edge-enhanced next-generation wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 5, pp. 1615–1631, 2022.

[6] H. Huo, J. Xu, G. Su, W. Xu, and N. Wang, "Intelligent MIMO detection using meta learning," *IEEE Wireless Communications Letters*, 2022.

[7] Z. Yin, W. Xu, R. Xie, S. Zhang, D. W. K. Ng, and X. You, "Deep CSI compression for massive MIMO: A self-information model-driven neural network," *IEEE Transactions on Wireless Communications*, 2022.

[8] J. Zhao, X. Chang, Y. Feng, C. H. Liu, and N. Liu, "Participant selection for federated learning with heterogeneous data in intelligent transport system," *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[9] R. Xie, W. Xu, Y. Chen, J. Yu, A. Hu, D. W. K. Ng, and A. L. Swindlehurst, "A generalizable model-and-data driven approach for open-set RFF authentication," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4435–4450, 2021.

[10] Y. Qian, J. Xu, S. Zhu, W. Xu, L. Fan, and G. K. Karagiannidis, "Learning to optimize resource assignment for task offloading in mobile edge computing," *IEEE Communications Letters*, 2022.

[11] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, "An energy efficient IoT data compression approach for edge machine learning," *Future Generation Computer Systems*, vol. 96, pp. 168–175, 2019.

[12] M. Chen, Z. Yang, W. Saad, C. Yin, H. V. Poor, and S. Cui, "A joint learning and communications framework for federated learning over wireless networks," *IEEE Transactions on Wireless Communications*, vol. 20, no. 1, pp. 269–283, 2020.

[13] Z. Yang, M. Chen, W. Saad, C. S. Hong, and M. Shikh-Bahaei, "Energy efficient federated learning over wireless communication networks," *IEEE Transactions on Wireless Communications*, vol. 20, no. 3, pp. 1935–1949, 2020.

[14] H. Kimm, I. Paik, and H. Kimm, "Performance comparison of TPU, GPU, CPU on Google Colaboratory over distributed deep learning," in *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, 2021, pp. 312–319.

[15] M. Malinowski, G. Swirszcz, J. Carreira, and V. Patraucean, "Side-ways: Depth-parallel training of video models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[16] S. Park and T. Suh, "Speculative backpropagation for CNN parallel training," *IEEE Access*, vol. 8, pp. 365–374, 2020.

[17] Z. Huo, B. Gu, H. Huang *et al.*, "Decoupled parallel backpropagation with convergence guarantee," in *International Conference on Machine Learning*. PMLR, 2018, pp. 2098–2106.

[18] S. Wang, Y. Bai, and G. Pekhimenko, "BPPSA: Scaling back-propagation by parallel scan algorithm," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 451–469, 2020.

[19] A. Nøkland, "Direct feedback alignment provides learning in deep neural networks," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[20] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan *et al.*, "Towards federated learning at scale: System design," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 374–388, 2019.

[21] X. Lu, Y. Liao, P. Lio, and P. Hui, "Privacy-preserving asynchronous federated learning mechanism for edge network computing," *IEEE Access*, vol. 8, pp. 48970–48981, 2020.

[22] S. Luo, X. Chen, Q. Wu, Z. Zhou, and S. Yu, "HFEL: Joint edge association and resource allocation for cost-efficient hierarchical federated edge learning," *IEEE Transactions on Wireless Communications*, vol. 19, no. 10, pp. 6535–6548, 2020.

[23] Q. Zeng, Y. Du, and K. Huang, "Wirelessly powered federated edge learning: Optimal tradeoffs between convergence and power transfer," *IEEE Transactions on Wireless Communications*, vol. 21, no. 1, pp. 680–695, 2022.

[24] D. Chen, Y.-C. Liu, B. Kim, J. Xie, C. S. Hong, and Z. Han, "Edge computing resources reservation in vehicular networks: A meta-learning approach," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 5, pp. 5634–5646, 2020.

[25] J. Zhao, R. Han, Y. Yang, B. Catterall, C. H. Liu, L. Y. Chen, R. Mortier, J. Crowcroft, and L. Wang, "Federated learning with heterogeneity-aware probabilistic synchronous parallel on edge," *IEEE Transactions on Services Computing*, 2021.

[26] J. Park, S. Samarakoon, A. Elgabli, J. Kim, M. Bennis, S.-L. Kim, and M. Debbah, "Communication-efficient and distributed learning over wireless networks: Principles and applications," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 796–819, 2021.

[27] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1085–1101, 2020.

[28] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*, 2014.

[29] A. Sergeev and M. D. Balso, "Horovod : fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799v3*, no. September, 2017. [Online]. Available: <http://arxiv.org/abs/1802.05799v3>

[30] J. Wang, A. K. Sahu, Z. Yang, G. Joshi, and S. Kar, "MATCHA: Speeding Up Decentralized SGD via Matching Decomposition Sampling," *2019 6th Indian Control Conference, ICC 2019 - Proceedings*, pp. 299–300, 2019.

[31] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, pp. 1–43, 2018.

[32] C. Elliott, "The simple essence of automatic differentiation," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, jul 2018.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[34] S. Zhang, P. Gunupudi, and Q.-J. Zhang, "Parallel back-propagation neural network training technique using CUDA on multiple GPUs," in *2015 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, 2015, pp. 1–3.

[35] Y. Liu, W. Jing, and L. Xu, "Parallelizing backpropagation neural network using MapReduce and cascading model," *Computational Intelligence and Neuroscience*, vol. 2016, 2016.

[36] J. Launay, I. Poli, F. Boniface, and F. Krzakala, "Direct feedback alignment scales to modern deep learning tasks and architectures,"

Advances in Neural Information Processing Systems, vol. 33, pp. 9346–9360, 2020.

- [37] F. Wang, G. Chen, W. Zhang, and T. Rompf, "Parallel training via computation graph transformation," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3430–3439.
- [38] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *arXiv preprint arXiv:1512.00567*, vol. abs/1512.00567, 2015. [Online]. Available: <http://arxiv.org/abs/1512.00567>
- [39] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2012, pp. 99–113.
- [40] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.
- [41] L. Wang, "Owl: OCaml Scientific Computing," <https://github.com/owlbarn/owl>, 2022, [Online; accessed 4-Sep-2022].
- [42] R. Sethi, "Complete register allocation problems," *SIAM Journal on Computing*, vol. 4, no. 3, pp. 226–248, 1975.
- [43] L. Wang, J. Zhao, and R. Mortier, *OCaml Scientific Computing*. Springer Cham, 2022.
- [44] D. Zennaro, A. Ahmad, L. Vangelista, E. Serpedin, H. Nounou, and M. Nounou, "Network-wide clock synchronization via message passing with exponentially distributed link delays," *IEEE Transactions on Communications*, vol. 61, no. 5, pp. 2012–2024, 2013.
- [45] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2961–2969.



Hao Tao received the B.E. and Ph.D. degrees in control science and engineering from Xi'an Jiaotong University, Xi'an, China, in 2009 and 2015, respectively. He joined China Ship Development and Design Center in 2015, and his current research interests are in signal processing, artificial intelligence, adaptive control, and robotics.



Liang Wang received the B.Eng. degree in computer science and mathematics from Tongji University, Shanghai, China, in 2003, the M.Sc. and Ph.D. degrees in computer science from the University of Helsinki, Finland, in 2011 and 2015, respectively. He is currently a Research Associate with the Computer Laboratory, University of Cambridge, U.K. His research interests include system and network optimization, modelling and analysis of complex networks, information-centric networks, and distributed data processing.



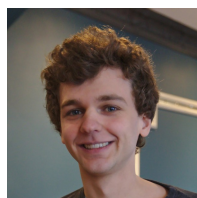
Jianxin Zhao received the Bachelor and Master degree in Software Engineering from the Beijing Institute of Technology, Beijing, China, in 2013 and 2015 respectively. He received his PhD degree in Computer Science from the University of Cambridge (affiliated with Corpus Christi College), supervised by Professor Jon Crowcroft. He is interested in various research topics, including numerical computation, high performance computing, machine learning, and their application in the real world. Currently he is working as a post-doc in the Beijing Institute of Technology, supervised by Prof. Chi H. Liu.

of Technology, supervised by Prof. Chi H. Liu.



Chi Harold Liu (SM'15) receives the Ph.D. degree from Imperial College, UK in 2010, and the B.Eng. degree from Tsinghua University, China in 2006. He is currently a Full Professor and Vice Dean at the School of Computer Science and Technology, Beijing Institute of Technology, China, and also the Director of Beijing Key Laboratory of Intelligent Information Processing. Before moving to academia, he worked for IBM Research - China, Deutsche Telekom Laboratories, Germany, IBM T. J. Watson Research Center, USA. His current research interests include the Mobile Crowdsensing and Deep Reinforcement Learning. He received the ACM SigKDD'21 Best Paper Award - Runner up and ACM MobiCom'21 Best Community Paper Runner-up Award. He serves as the Associate Editor for IEEE Transactions and Network Science and Engineering. He is a Fellow of IET, British Computer Society and Royal Society of Arts.

include the Mobile Crowdsensing and Deep Reinforcement Learning. He received the ACM SigKDD'21 Best Paper Award - Runner up and ACM MobiCom'21 Best Community Paper Runner-up Award. He serves as the Associate Editor for IEEE Transactions and Network Science and Engineering. He is a Fellow of IET, British Computer Society and Royal Society of Arts.



Pierre Vandenhove is a PhD student cosupervised by Mickael Randour from the Effective Mathematics Team at the University of Mons and by Patricia Bouyer-Decitre from the Laboratoire Méthodes Formelles at the École Normale Supérieure Paris-Saclay. His research interests lie in the game-theoretic approach to formal verification and in hybrid models. He obtained his master's degree from the University of Mons.



Jon Crowcroft has been the Marconi Professor of Communications Systems in the Computer Laboratory since October 2001. He has worked in the area of Internet support for multimedia communications for over 30 years. Three main topics of interest have been scalable multicast routing, practical approaches to traffic management, and the design of deployable end-to-end protocols. Current active research areas are Opportunistic Communications, Social Networks, Privacy Preserving Analytics, and techniques and algorithms to scale infrastructure-free mobile systems. He leans towards a "build and learn" paradigm for research. From 2016-2018, he was Programme Chair at the Turing, the UK's national Data Science and AI Institute, and is now researcher-at-large there. He graduated in Physics from Trinity College, University of Cambridge in 1979, gained an MSc in Computing in 1981 and PhD in 1993, both from UCL. He is a Fellow the Royal Society, a Fellow of the ACM, a Fellow of the British Computer Society, a Fellow of the IET and the Royal Academy of Engineering and a Fellow of the IEEE.

He graduated in Physics from Trinity College, University of Cambridge in 1979, gained an MSc in Computing in 1981 and PhD in 1993, both from UCL. He is a Fellow the Royal Society, a Fellow of the ACM, a Fellow of the British Computer Society, a Fellow of the IET and the Royal Academy of Engineering and a Fellow of the IEEE.



Peng Xu is currently a part-time PhD student at School of Computer Science and Engineering, Beijing Institute of Technology, China, under the supervision of Prof. Chi Harold Liu. He received his M.Eng. degree at Beijing University of Posts and Telecommunications, and his current research interests are edge computing and deep learning.