EdgePipe: Tailoring Pipeline Parallelism With Deep Neural Networks for Volatile Wireless Edge Devices

JinYi Yoon, Yeongsin Byeon, Jeewoon Kim, and HyungJune Lee, *Member, IEEE*

***Abstract*—As intelligence recently moves to the edge to tackle the problems of privacy, scalability, and network bandwidth in the centralized intelligence, it is necessary to construct an efficient yet robust deep learning model viable at edge devices, which are usually volatile in wireless links and device functionality. The intensive computation burden for deep learning at the edge side necessitates some level of parallel processing via acceleration. We propose *EdgePipe*, a deep learning framework based on deep neural networks (DNNs) with a mixture of model parallelism and pipeline training for high resource utilization over volatile wireless edge devices. To tackle the volatility problem in wireless links and device functionality, a concept of *super neuron* is defined to be a group of neurons across adjacent layers, which is the basis of model partitioning at edge devices. The relatively loss-resilient neuron structure prevents the entire forward or backward training paths from being totally broken down due to only some intermittent link or device failure caused by one or few devices. Furthermore, we design a subsequent pipeline training mechanism based on the prior super-neuron-based model partitioning for fast convergence with more training data in a fixed timeline. The experimental results have demonstrated that *EdgePipe* outperforms several counterpart algorithms including *PipeDream* under the volatile wireless lossy or device malfunctioning environments, while preserving the low interlayer communication overhead.**

***Index Terms*—Distributed deep learning, edge device, model parallelism, pipeline parallelism, volatile wireless links.**

I. Introduction

Deep neural networks (DNNs) have actively investigated for use in a wide range of application areas, such as classification, image recognition, and natural language processing. The size of neural networks has increased to a total of 10–20 million parameters, necessitating the use of significant amounts of computational resources, and the use of approaches, such as parallel processing via multilevel accelerators to facilitate efficient learning.

To make deep learning scalable in a distributed setting, parallelized DNN training can be used. This approach involves dividing the training data or the neural networks into subdata shards or submodel shards, respectively, [1], [2]. Data parallelism enables simultaneous learning using multiple processes, by replicating a neural network, leading to an acceleration of processing time caused by processing different data shards on different servers or GPUs in parallel [3], [4]. However, this approach suffers from the requirement for ongoing communication among the replicated models to synchronize the model parameters in a central server [5], [6]. Some researchers have investigated algorithms for model parallelism that partitions a deep model into multiple submodels and allocates these models to different servers or GPUs [7], [8]. Although this approach reduces storage, computation, and communication overhead, the sequence of cross-device computations in the forward and backward passes should be performed in order, and still incurs a communication overhead. Since model parallelism may cause problems of resource underutilization, this approach parallelism does not necessarily guarantee improvement of training efficiency.

To resolve the resource underutilization problem, pipeline parallelism is introduced to facilitate the fast training of DNNs [9]–[13]. Each forward or backward pass is considered to be a single pipeline stage, an approach which enables multiple devices to function concurrently. Although these parallel approaches are designed to run large-scale DNNs based on powerful servers or GPUs in a distributed manner, the issue of general, yet more challenging, edge intelligence has not been discussed in detail [14], [15]. Edge devices tend to be resource constrained and communication-fragile due to the low-power wireless properties of transmission and reception, so the edge computing environment makes parallelism in deep learning challenging [16], [17]. The existing pipeline parallelism has been mostly explored assuming the existence of wired interconnection units, such as servers or GPUs. Unreliable network connectivity due to, for example, the lack of some required transport layer support can incur substantial overhead at the edge level, and can lead to learning instability and the generation of invalid inferences. Model parallelism in deep learning at the edge therefore poses a new challenge: to develop a volatility-resilient parallel structure at the level of a network of resource-constrained edge devices [18], [19].

In this article, we propose *EdgePipe*, a hybrid parallelized deep learning framework based on model parallelism with pipeline acceleration over volatile edge networks. As a hybrid parallelism, we aimed to achieve stable yet cost-effective learning at fragile wireless networks without any transport layer support, through salient partitioning in the level of neurons across layers, as shown in Fig. 1. Each device is allocated a set of selective neurons, which should be resilient against volatility, and computes their activity in forward and

Manuscript received September 3, 2021; accepted November 19, 2021. Date of publication November 30, 2021; date of current version July 7, 2022. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) under Grant NRF-2021R1A2B5B01002906. *(Corresponding author: HyungJune Lee.)*

The authors are with the Department of Computer Science and Engineering, Ewha Womans University, Seoul 03760, South Korea (e-mail: hyungjune.lee@ewha.ac.kr).

Digital Object Identifier 10.1109/JIOT.2021.3131407

2327-4662 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

backward passes. On top of model parallelism, we further leveraged pipeline parallelism, and reduced the training time at the edge by increasing the utilization of multiple workers (i.e., devices).

To identify a relatively loss-resilient neuron structure, we introduce the concept of a *super neuron*, which is a basis for allocation of edge devices to DNNs. A model is partitioned into partial submodels to the edge, based on the super neuron by discovering the optimal mapping between the device-to-device communication network topology and super-neuron-based neural network topology. After the neuron-to-device allocation, we accelerate the training procedure via pipeline scheduling that computes the forward and backward passes alternately in an efficient, distributed manner. Since a series of subsequent data can be inserted and processed before a previous data item is fully processed at the end of the backward pass, *EdgePipe* complements the unstable intermittent interconnectivity by engaging devices based on a super-neuron-based network structure, with substantial reduction in training time.

To the best of our knowledge, this is the first paper to tackle the problem of pipelined model parallelism with DNNs in volatile wireless edge devices. We demonstrated that *EdgePipe* is a feasible approach to distributed deep learning under lossy wireless environments. The main contributions of this work can be summarized as follows.

1) We present an efficient neuron-to-device allocation based on the concept of a *super neuron* that is a resilient partitioning structure for edge DNNs against unreliable wireless dynamics.
2) We adopt a pipeline approach to concurrently process a series of minibatch data within the edge network, achieving high accuracy, while reducing the training time, even when undergoing communication failures.
3) Based on extensive experiments, we have demonstrated that *EdgePipe* outperforms several important counterpart algorithms including *PipeDream*, in volatile wireless lossy or device malfunctioning environments, 2.47 times faster than *Model Parallelism* with 2.13 times higher inference than the core mechanism obtained from *PipeDream* [20].

II. Related Work

Our work is a part of federated learning, to a novel problem space, which is wireless lossy networks of edge devices. To complement the innately slow training at the edge due to resource or communication volatility and constraints, our approach is applied with pipeline acceleration.

A. Federated Learning

To process deep learning on multiple devices, training data are split into multiple subtasks using data parallelism [1]. Data parallelism is the process of replicating the network model on multiple devices, where each device trains a subset of the training data [21]–[23]. Some researchers have exploited task parallelism with data parallelism, which is optimized for use with memory parallel computers [24]. *Accelerator* [25] solves the problem of general-purpose usages of GPUs by applying data parallelism. Since data parallelism involves holding a copy of the complete network model, it can be applied to any deep learning architecture. As the neural networks become large, a single device cannot store the full network model. The model parameters need to be synchronized among devices, causing heavy communication overhead. As the number of training devices increases to accelerate learning, the communication overhead becomes a critical issue. Therefore, to overcome the problem of synchronization, some researchers have investigated parameter transfer management [4]–[6]. *AdaComp* [26] introduces data parallelism with a parameter server into edge–device networks.

To solve the problem of the feasibility of large-scale models at the edge, model parallelism has been studied by partitioning a learning model among devices [8], [27]. Since the memory footprint can be reduced due to model partitioning, model parallelism enables large-scale training on resource-constrained edge devices [28]. Some heuristic algorithms have used partitioning algorithms based on computational graphs in *TensorFlow* [29], [30]. However, in model parallelism, since a forward pass for new data can start to be processed after the backward steps of its precedent training batch are completed, only one device works at a time, and it cannot contribute to training speedup. To accelerate model parallelism, *Layerwise Staleness* and *DSP* [7] have been tried, to solve the straggler issue by training each partitioned model independently. Large amounts of communication are needed among training devices to share the intermediate computational results. Recent work has combined model parallelism and data parallelism, in an approach called hybrid parallelism [31]–[34].

B. Pipeline Acceleration

Although data and model parallelism approaches aim to facilitate the learning of large networks over end devices, they are still not able to fully utilize the given resources. By borrowing ideas from pipelines, some researchers have leveraged learning acceleration into model parallelism [9], [10], [12], [13]. This approach enables fast learning, in which the devices are efficiently utilized, while each device holds only a subset of the model. *PipeDream* [20] brings the pipeline framework into deep learning passes in practice. This approach partitions the layers between multiple GPUs, and allows each device to process one forward and one backward pass repeatedly. By designing a series of forward and backward passes to overlap over the devices, some limited computational resources can be optimally utilized. To solve the issue of parameter staleness which arises in *PipeDream*, *SpecTrain* [35] predicts future weights, maintaining rapid learning. Recently, *GPipe* [36] has facilitated faster learning by reducing idle time by splitting minibatches into microbatches. *HetPipe* [37] divides GPUs into a group of virtual workers by applying both model and data parallelism. The intravirtual workers process pipelined model parallelism, while the intervirtual workers parallelize the training data. To facilitate learning, *XPipe* [11] separates data into finer batches, and minibatches are split into microbatches.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

However, these parallelism methodologies rely on the existence of a reasonably stable network connection among the computing devices and internal GPUs. As on-device learning is more common with edge devices, wireless connectivity poses a new, challenging but interesting, problem. More closely related to our problem, edge computing has been widely investigated in mobile-edge offloading [38]–[40]. Some researchers have considered DNN partitioning to optimize resources across edge servers for AI-based applications in the user equipments [41]. Although these mobile-edge devices are supposed to be wirelessly connected, the innate instability of the wireless link connection has not been considered to date. The existing approaches to on-device learning have focused only on the wireless communication between the parameter server and the end devices with the complete network model [14], [42], [43]. In this study, we introduced a model-partitioning-based distributed learning approach using edge devices in a volatile wireless network, without maintaining any centralized parameter synchronization. We present a resilient partitioned model based on the concept of a *super neuron* and address the transmission failure problem along with pipeline acceleration.

III. System Model

We present a distributed deep learning scheme using multiple edge devices over wireless links. In a lossy network environment, not only the risk of managing the unstable neural networks, but also the peer-to-peer communication overhead should be considered in order to implement a distributed deep learning system. To tackle this problem, we suggest a simple yet efficient model partitioning approach to construct a wireless network-optimized DNN model.

We overcome the problem of resource underutilization in model parallelism and the lag in the learning process due to the possible connection or device loss caused by the inherent nature of edge devices, by applying pipeline scheduling. The hardware efficiency is enhanced by keeping the pipeline as full as possible. Thus, the learning model can be established earlier even under dynamic lossy networks. We address two key questions: 1) what is a desirable partitioning basis for edge devices considering the volatile network connectivity? and 2) how can the training of the neural networks, partitioned over the devices, be parallelized to achieve fast training or high throughput with more training data?

We assume that edge devices are wirelessly connected via a low-power wireless radio link, such as IEEE 802.11 or 802.15.4, while not using any centralized Cloud access. As our work is based on model parallelism, all of the data sets are assumed to be independent and identically distributed (i.i.d).

We aimed to reduce the training time by overlapping the computation of multiple resources, while constructing a desired learning model. From the perspective of edge networks, the model is designed to alleviate in-flight data failure with reasonable interlayer communication. We assume that edge devices can communicate with other devices using a wireless radio such as IEEE 802.11, and all of the data sets are i.i.d.

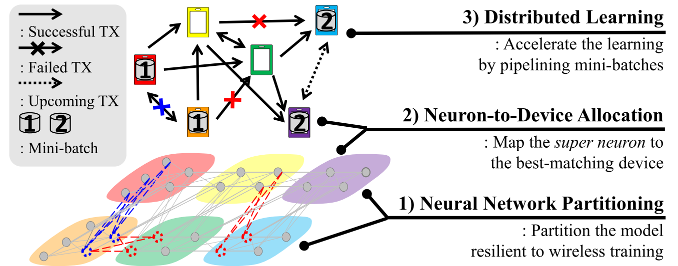


Fig. 1. System overview of *EdgePipe* that exploits pipeline parallelism for edge DNNs over a volatile wireless network.

Our distributed deep learning framework, *EdgePipe*, takes a hybrid approach to combining model parallelism and pipeline parallelism for scalable and *volatile* learning at the edge under some level of uncertainty in the device-to-device connection, as illustrated in Fig. 1. *EdgePipe* consists of two procedures, as follows: 1) model parallelism and 2) distributed learning using a pipeline.

1) *Model Parallelism:* To solve a DNN problem with a group of edge devices, we partition a DNN into partial subnetworks. By allocating a partial subnetwork of suitable interlayer neurons to a device, with neither purely vertical nor purely horizontal allocation, we prevent the neurons in a single layer from being grouped and processed with a single device, while also reducing the transmission overhead during training. The partial subnetworks are distributed to edge devices each finding its own best-matching device. At the initial setup phase, we select the best centrality node as a temporary device coordinator. The selected device coordinator profiles the neural network structure and performs model partitioning based on the partitioning basis of a *super neuron*, for constructing a volatility-resilient learning structure.
2) *Distributed Learning With Pipeline Scheduling:* Once the neurons are assigned to the edge devices, we incorporate a pipeline scheduling over the forward and backward passes. Once an edge device receives partial computational results from a neighboring device, which is in charge of a group of neurons located at the prior layer, it continues to compute the feed forward calculation in the forward pass, or the gradient in the backward pass, and sends it to a neighboring device in charge of the next layer.

IV. Model Parallelism

To tailor deep learning to distributed low-end edge devices, it is essential to reduce the computation and storage overhead at the edge, for sustainability. Particularly, in a volatile wireless network in which edge devices are usually formed using device-to-device network connections, it is important to understand a durable model partitioning structure that can prevent global breakdown in the learning paths due to intermittent link or device failures. We aim to derive a way to efficiently partition and allocate a neural network model, which is resilient and cost-effective in volatile edge networks.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

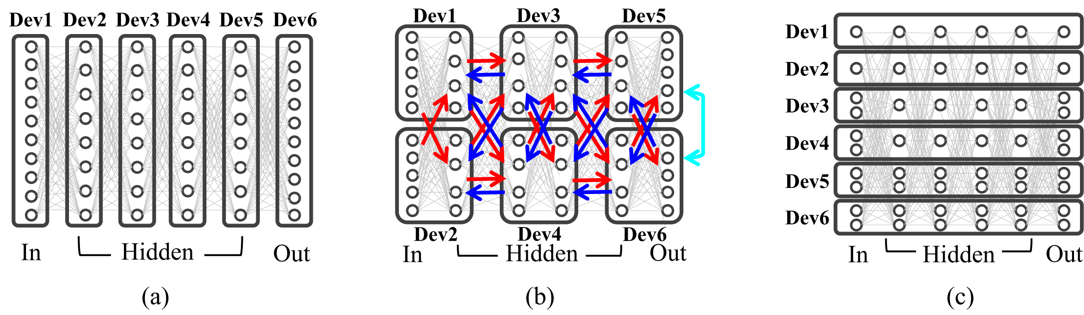


Fig. 2. Neural network partitioning into a group of *super neurons* to make a distributed DNN resilient to the uncertainty of the communication network. The red, blue, and cyan lines denote the forward, backward, and output sharing passes, respectively. (a) Vertical allocation. (b) Hybrid allocation. (c) Horizontal allocation.

As an initial partitioning setup, *EdgePipe* selects a device coordinator that is supposed to perform a model partitioning by taking into account the device-to-device connectivity status and the workloads at the devices. We choose a device with the highest network centrality measure as the coordinator in the distributed learning setup for the edge devices. Among various network centrality measures, the classic closeness centrality [44] based on the expected number of transmissions considering the packet reception rate, which has been recognized as a general metric with which to measure the link quality, is adopted. This is because the shortest-path-based decision result should be efficiently spread from the coordinator to the other devices in the wireless environment. After the coordinator discovers a suitable set of neurons, the coordinator informs all of the edge devices of the result of the preliminary configuration, indicating which parts of the neural networks should be partitioned and then allocated to a specific edge device.

A. Neural Network Partitioning

We suggest a cost-effective, training-stable partitioning scheme for lossy wireless networks. Under volatile wireless link dynamics, if all neurons from a single layer are allocated to one or a few devices, the layer can be entirely lost or drastically damaged, and the forward and backward learning in subsequent layers can be seriously impaired.

To reduce the layer-to-device dependency as far as possible, we introduce a concept of a *super neuron* (Fig. 2); a group of neurons operating across adjacent layers. The super neuron is designed to be a basis for neuron-to-device allocation of tasks. By grouping the neurons located in adjacent layers and assigning them to a device for a later distributed learning process, the cross-layer forward and backward learning passes can be viable within a single device, despite the volatility in wireless links. Super neurons acting across adjacent layers can contribute to reducing the interlayer communication overhead.

To form a set of super neurons in a DNN across L_{layers} layers, we propose a key parameter, M_{layers}, as the maximum number of contiguous layers in which the neurons in a super neuron are located, for a specific device. We classify the model partitioning into three categories with respect to M_{layers}, where $1 \leq M_{\text{layers}} \leq L_{\text{layers}}$ as follows.

1) *Vertical Allocation:* This type of allocation occurs when the maximum number of contiguous layers of which a device takes charge is met; that is, $M_{\text{layers}} = 1$. In *Vertical* allocation, each device is supposed to take charge of the neurons from a single layer, as shown in Fig. 2(a). In the worst case, the forward and backward learning paths in a DNN can be totally collapsed due to a complete loss at a single layer caused by communication failure to or from a responsible device. *PipeDream* [20], *GPipe* [36], and *GNMT* [45] belong to this category in which the communication volatility is not seriously considered because they have assumed an almost lossless GPU-to-GPU network environment.
2) *Hybrid Allocation: Hybrid* allocation takes place when the maximum number of contiguous layers per device is larger than 1, i.e., $1 < M_{\text{layers}} < L_{\text{layers}}$. In *Hybrid* allocation, each device takes in charge of the neurons across adjacent layers, with better continuity in the learning process, while reducing some possible unnecessary interlayer communication overheads, as shown in Fig. 2(b). *EdgePipe* uses the *Hybrid* allocation to construct a viable model partitioning in DNNs in volatile wireless networks.
3) *Horizontal Allocation:* In contrast to *Vertical* allocation, *Horizontal* allocation allows each device to be in charge of the neurons located over all of the layers of a DNN, where $M_{\text{layers}} = L_{\text{layers}}$, as in Fig. 2(c). If many devices are involved in forward and backward passes over unnecessarily long consecutive layers, the communication overhead can be significant. More seriously, since the computational results may be lost for many device-to-device communications in the middle of training, the computed gradients cannot be successfully transmitted to the next layer in the backward pass. Another critical drawback occurs when the horizontal allocation structure cannot benefit from pipeline acceleration, which will be discussed in the next section.

We present a simple yet efficient heuristic algorithm that partitions a two-phase hybrid process of dividing a model first horizontally into layers, and then vertically into neurons, as described in Algorithm 1. Given the constraints of the number of neurons at each layer, the maximum number of contiguous

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Model Partitioning

```
1: Function model-partitioning (L_layers, M_layers, N_neurons: # of
   neurons in neural network in list, N_devices: # of devices)
2:   if isDeviceCoordinator() == TRUE then
       // I. Split in the level of layers
3:     allocatedLayer[1 : ⌊2/N_devices⌋] = ⌊L_layers/⌊N_devices/2⌋⌋;
4:     allocatedLayer[end − (L_layers − M_layers × ⌊N_devices/2⌋) +
       1 : end]+ = 1;
       // II. Allocate # of devices in proportion to # of layers
5:     for idx = 1 : length(allocatedLayer)
6:       allocatedDevice[idx] = rounding(N_devices/L_layers×
         length(allocatedLayer[idx]));
7:     end for
       // III. Split in the level of neurons for each layer
8:     SN = [ ]; // allocated neurons
9:     for w = 1:length(allocatedLayer)
10:      h = 1;
11:      for l in allocatedLayer[w]
12:        n_neurons = N_neurons[l];
13:        n_devices = allocatedDevice[w];
14:        allocatedNeuron
             = task-division(n_neurons, n_neurons/n_devices);
15:        SN[w][h].append([l, allocatedNeuron]);
16:        h = h + 1;
17:      end for
18:    end for
       // IV. Map device topology and super neuron topology
19:    deviceOrder = device-mapping(SN, N_devices);
20:    Assign SN[w][h] to deviceOrder[idx];
21:  else
22:    Receive the task from deviceCoordinator;
23:  end if
24: end Function
25: Function task = task-division (n_task: # of tasks, M_task: maximum
    # of tasks)
26:    task = [ ]; // list of allocated layers
27:    n = n_task;
28:    while n > 0
29:      if n ≥ M_task then
30:        task.appendleft([n − M_task + 1:M_task]);
31:        n = n - M_task;
32:      else
33:        task.appendleft([1:n]);
34:        n = n − n;
35:      end if
36:    end while
37: end Function
```

layers per device M_{layers} and the number of whole layers L_{layers}, the model can be split initially by grouping from the output layer as follows:

$$M_{\text{layers}} = \lfloor L_{\text{layers}} / \lfloor N_{\text{devices}}/2 \rfloor \rfloor \tag{1}$$

where N_{devices} is the number of devices. The complete network is divided into $\lfloor N_{\text{devices}}/2 \rfloor$ of M_{layers} layers. The remaining layers $L_{\text{layers}} - M_{\text{layers}} \times \lfloor N_{\text{devices}}/2 \rfloor$ are equally assigned one by one (since the number of the remaining layers is always smaller than $\lfloor N_{\text{devices}}/2 \rfloor$). If there are a sufficient number of devices ($N_{\text{devices}} \geq 2 \times L_{\text{layers}}$) that multiple devices can take charge of a single layer together, *Hybrid* allocation becomes *Vertical* allocation, which can be accelerated by pipelining. In the case of two or three devices ($M_{\text{layers}} = L_{\text{layers}} \Leftrightarrow \lfloor N_{\text{devices}}/2 \rfloor = 1$), *Hybrid* allocation becomes horizontal allocation, that allows a single device to take charge of neurons across layers.

In Fig. 2(b), we partition a model with six layers into six devices by allowing a maximum of two layers to a device, where $N_{\text{neurons}} = [10, 8, 8, 8, 8, 10]$, $L_{\text{layers}} = 6$, $N_{\text{devices}} = 6$, and $M_{\text{layers}} = \lfloor 6/\lfloor 6/2 \rfloor \rfloor = 2$ with the hybrid allocation algorithm. First, the model is split into 2, 2, and 2 layers (*allocatedLayer* = [[1, 2], [3, 4], [5, 6]]) at the level of layers. Then, we assign the number of devices in proportion to the number of layers, meaning that two devices are allowed to process up to two layers, (*allocatedDevice* = [2, 2, 2]). Finally, we partition at the level of neurons equal to the number of devices, for task balancing.

To make DNN scalable for low-performing edge devices, our partitioning scheme contributes to reducing the load of *dense* layers, which accounts for most of the computation time [46]. Other types of layers, such as convolution layers or pooling layers, which generally lead to high performance, can be leveraged based on a layerwise allocation in the convolutional neural network (CNN). Our work mainly focuses on dense layers, which are computationally intensive, but can feasibly be stacked in most DNNs, including CNNs.

B. Neuron-to-Device Allocation

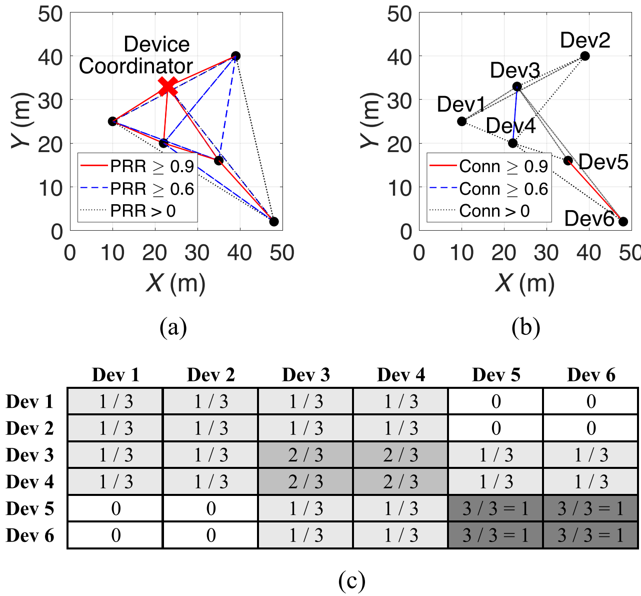
Once a DNN model is partitioned into a set of super neurons, the number of which is the same as the number of devices, we solve the problem of super neuron-to-device mapping considering volatile device-to-device wireless dynamics.

In a general setting, some super neurons need to communicate with each other frequently, whereas other pairs need only intermittent or almost no communication. A salient mapping decision for super neuron-to-device may dominate the success or failure when learning a DNN model operating under volatile wireless links.

First, we calculate the relative connectivity ratio of a specific super neuron-to-super neuron link out of all possible pairs and form a directed graph with the weight of the relative importance ratio in the range [0, 1]. For example, in Fig. 2(b), the total number of training passes from Device 3 to Device 4 is 2; one from the forward pass and one from the backward pass. Out of all possible links, the maximum number of training passes is three, at the logical links of Device 5-to-Device 6 and Device 6-to-Device 5 for one forward pass, one backward pass, and the result sharing pass at the output layer. To give a higher priority to a super neuron link with higher connectivity to another super neuron, we normalize each weight with the maximum number of training passes out of all possible links. The calculated exemplary super neuron topology is provided in Fig. 3(c).

Then, given a device-to-device network topology [Fig. 3(a)] and the calculated super neuron topology [Fig. 3(c)], we finally identify the optimal mapping between the device-device network topology and the super neuron topology. We implement a simple genetic algorithm that determines the minimum edit distance between two topologies, as in Algorithm 2. The network topology *devNet* is presented with a directed graph based on the packet reception ratio (PRR) between devices in

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.



	Dev 1	Dev 2	Dev 3	Dev 4	Dev 5	Dev 6
Dev 1	1 / 3	1 / 3	1 / 3	1 / 3	0	0
Dev 2	1 / 3	1 / 3	1 / 3	1 / 3	0	0
Dev 3	1 / 3	1 / 3	2 / 3	2 / 3	1 / 3	1 / 3
Dev 4	1 / 3	1 / 3	2 / 3	2 / 3	1 / 3	1 / 3
Dev 5	0	0	1 / 3	1 / 3	3 / 3 = 1	3 / 3 = 1
Dev 6	0	0	1 / 3	1 / 3	3 / 3 = 1	3 / 3 = 1

(c)

Fig. 3. Neuron-to-device allocation for a neural network with six layers using six devices spread over 50×50 m^2. (a) Device-to-device network topology based on PRR. (b) Super neuron-to-device mapping after allocation. (c) Relative connectivity of super neuron network topology.

the range of [0, 1], while the super neuron topology *SNNet* is described with another directed graph based on the relative connectivity ratio between super neurons in the range of [0, 1]. We continue to compute an elementwise product based on the Hadamard product [47] between two directed graphs by reordering the device sequence and select the shorter distance over generations. Finally, a topology mapping with the largest elementwise sum of the Hadamard product is reached, as illustrated in Fig. 3(b).

V. Acceleration of Distributed Learning

Our proposed model parallelism allows edge devices operating over volatile wireless links to train a DNN by partitioning it into partial submodels based on a new neuron group structure, a *super neuron*, which is resilient to the innate link volatility. However, only with the model parallelism, the distributed neural network may converge slowly, due to the constraint of limited computational resources at the edge and also some possible training losses caused by link losses between the devices.

In a distributed learning scenario, as long as an edge device releases its local forward or backward computation involved with its allocated super neuron for a certain minibatch data, it can process the incoming computation for the next minibatch data. This approach means that, if a well-designed learning schedule at the level of each device and its entire device group is ready, there exists the opportunity for multiple devices to process the learning procedures at the same time, via parallelism.

We apply a pipeline mechanism to the super-neuron-based model partitioning for wireless DNN training, in order to significantly reduce the training time. Each worker (i.e., device) arranges its own schedule according to the allocated layers,

Algorithm 2 Neuron-to-Device Mapping

1: **Function** *topology-mapping* ($N_{devices}$: # of devices, *SNNet*: super neuron topology, *devNet*: device-to-device network topology)
  // I. Get the initial population of element-wise sum of Hadamard Product
2: *bestNet = devNet*;
3: *bestHP = element-sum*(*SNNet* ∘ *bestNet*);
  // II. Reproduct the next population
4: **for** *generation* times
  // II-A. Randomly reorder the network
5: *randOrder = random-permutation*($N_{devices}$);
6: *randNet = reorder-nodes*(*bestNet*, *randOrder*);
7: *randHP = element-sum*(*SNNet* ∘ *randNet*);
  // II-B. Swipe the network
8: *swipeOrder* = [2:$N_{devices}$, 1];
9: *swipeNet = reorder-nodes*(*bestNet*, *swipeOrder*);
10: *swipeHP = element-sum*(*SNNet* ∘ *swipeNet*);
  // II-C. Find the maximum summation of elements in Hadamard product of device network and super neuron network
11: **if** *randHP == max* (*randHP*, *swipeHP*, *bestHP*) **then**
12: *bestNet = randNet*;
13: *bestHP = randHP*;
14: **else if** *swipeHP == max* (*randHP*, *swipeHP*, *bestHP*) **then**
15: *bestNet = swipeNet*;
16: *bestHP = swipeHP*;
17: **end if**
18: **end for**
19: *devNet = bestNet*;
20: **end Function**

by alternating the forward and backward passes. During the pipeline training process, when a device receives intermediate computation results from the one or more devices in charge of its prior layer, it processes the forward or backward computation. Then, it broadcasts the updated result to the devices in charge of its next layer.

A. Pipeline Scheduling

We train a series of minibatch data in parallel by borrowing a pipeline scheduling idea from *PipeDream* [20]. Let us define the forward computation of layer j for the ith data as FW_j^i and the backward as BW_j^i. For example, the training paths for data 10 in a vanilla neural network with three layers can be represented as $[\mathrm{FW}_1^{10} \rightarrow \mathrm{FW}_2^{10} \rightarrow \mathrm{FW}_3^{10} \rightarrow \mathrm{BW}_3^{10} \rightarrow \mathrm{BW}_2^{10}]$. It should be noted that at BW_1^{10}, there is no backward computation needed in the input layer. Then, a device alternates the forward passes over the allocated layers in ascending order and the backward passes in descending order, for data i where $i = 1, 2, \ldots,$ as follows:

$$\mathrm{FW}_{l_1}^i \rightarrow \mathrm{FW}_{l_2}^i \rightarrow \cdots \rightarrow \mathrm{FW}_{l_{w_n}}^i \rightarrow \mathrm{BW}_{l_{w_n}}^{i-P+w} \rightarrow \mathrm{BW}_{l_{w_n-1}}^{i-P+w} \rightarrow \cdots \rightarrow \mathrm{BW}_{l_1}^{i-P+w} \quad (2)$$

over the contiguous timeslots soon after data i starts to be processed in the forward pass, where $l_1, \ldots, l_{w_n} \in$ *allocatedLayer*$[w]$ in the ascending order, P is the number of stages or layers in the super-neuron-based network for the device mapped with $SN[w, h]$ at the wth layer of the super-neuron-based network, and the hth $(1 \leq h \leq H)$ horizontal super neuron at the wth layer in the network, under

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

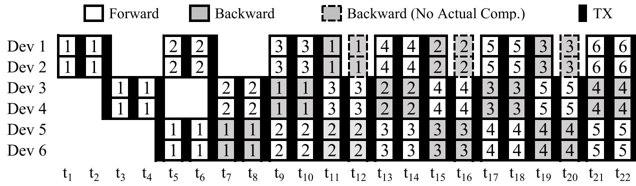


Fig. 4. Pipeline scheduling with an example of a super-neuron-based partitioned network in which the number denotes the data ID i.

the relationship of $L_{\text{layers}} = P \cdot M_{\text{layers}}$. The case of $\text{BW}_{l_n}^{i-P+w}$ where $i - P + w < 1$ is considered to be a pipeline stall. To use the parameters as late as possible, a device mapped with $SN[w, h]$ has $\Sigma_{idx=1}^{w-1} length(allocatedLayer[idx])$ stalls at the beginning. As another stall case, since there is no computation required at the input layer during the backward passes, one stall occurs at the last backward pass of BW_1^i. For example, as in Fig. 2(b), Device 4, which is located at $SN[2, 2]$ in the super-neuron-based network, takes in charge of the layers of {3, 4}, and the pipeline schedule starts with $\Sigma_{idx=1}^{w-1} length(allocatedLayer[idx]) = 2$ stalls, where the values of P and M_{layers} are 3 and 2, respectively. Then, it proceeds to perform the forward and backward computations, according to (2) with a sequence of [$\texttt{stall} \rightarrow \texttt{stall} \rightarrow \text{FW}_3^1 \rightarrow \text{FW}_4^1 \rightarrow \text{BW}_4^0\ (\texttt{stall}) \rightarrow \text{BW}_3^0\ (\texttt{stall}) \rightarrow \text{FW}_3^2 \rightarrow \text{FW}_4^2 \rightarrow \text{BW}_4^1 \rightarrow \text{BW}_3^1 \rightarrow \cdots$], as also illustrated in Fig. 4.

Theorem 1 (Acceleration Time Complexity): To process N_{data} minibatch data with a neural network of L_{layers} layers with the super neuron allocation with M_{layers}, *EdgePipe* requires a total number of timeslots of

$$2L_{\text{layers}} + 2M_{\text{layers}}(N_{\text{data}} - 1) - n_{cut} = O\big(N_{\text{data}} \cdot M_{\text{layers}}\big)$$

where n_{cut} is 1 for nonhorizontal allocations and N_{data} for the horizontal allocation.

Proof: For nonhorizontal allocations, the last stage in the input layer for the backward pass does not need to be computed. The processing latency for the first minibatch data is L_{layers} for the forward pass and $L_{\text{layers}} - 1$ for the backward pass. From that time on, the nonhorizontal allocations need additional timeslots with $2 \cdot M_{\text{layers}}$ to process each additional minibatch data. For the *Horizontal*, allocation which cannot utilize any pipelining schedule since one device should involve all of the computation over all of the layers, the processing latency for each minibatch data is given by $2L_{\text{layers}} - 1$, and the total number of timeslots is given by $(2L_{\text{layers}} - 1) \cdot N_{\text{data}}$. ■

Theorem 2 (Acceleration by Super Neuron Structure): The training time in a distributed DNN can be reduced via a pipeline schedule with a factor of $O(L_{\text{layers}}/M_{\text{layers}}) = O(P)$, where P is the number of pipeline stages in a super-neuron-based network.

Proof: Given a fixed number of minibatch data for training, the total learning time is given by $O(K \cdot M_{\text{layers}}) = O(K' \cdot M_{\text{layers}}/N_{\text{layers}})$. The reduction ratio in time is given by $O(1/(K' \cdot M_{\text{layers}}/N_{\text{layers}})) = O(K'' \cdot N_{\text{layers}}/M_{\text{layers}}) = O(K'' \cdot P)$ where $N_{\text{layers}} = P \cdot M_{\text{layers}}$. The distributed DNN training can be accelerated according to the number of pipeline stages in the super-neuron-based network. ■

B. Distributed Learning Over Volatile Links

The training data are split into a set of minibatch data. Each minibatch data is injected from the input layer controlled by a specific device or devices. In the forward pass, each device computes the intermediate results and propagates them to the devices in charge of the next layer. In the backward pass, the devices compute the gradients and pass them to the devices in charge of the preceding layer. For the device-to-device communication, each device broadcasts a packet in which the intermediate results are embedded in the packet payload, and the device and data IDs are stored in the packet header.

In a lossy wireless environment, the layer-to-layer transmission over the forward and backward passes can be lost. Given a calculated pipeline schedule, each device receives the results by a certain time, starts its subsequent computation, and shares it with surrounding devices by broadcast. Based on the pipelined schedule, if the next target data arrive, the device stops waiting and processes the current task. When some partial information has been received from a certain device, the corresponding information is treated as null, and the process is continued.

Since multiple minibatch data are being processed in the pipeline scheduling, correct decisions about which parameters are to be used and updated is required. We design each device to use the most recent parameters for the forward pass, and reflect the calculated gradients to the most recent parameters for the backward pass, similar to [20]. The weight parameter for data i, $W[i]$ is calculated as follows:

$$W[i] = W[i-1] - \gamma \nabla f(W[i - (P - w + 1)]). \quad (3)$$

For example, in Fig. 4, when Device 4 is about to perform the calculation for data 4 in the forward pass, it uses the most recent weight, which has just been updated after backpropagating for data 2. Then, the computed gradients are used to update the weight parameter at data 3.

During backpropagation, a device responsible for a certain layer is able to calculate the gradients only if it receives all of the losses from its next layer. This means that if there is any transmission failure in its preceding backward pass, we cannot update all of its subsequent backpropagation. For this reason, *Horizontal* allocation has a high risk of failure in learning since it requires many device-to-device communications between two adjacent layers. The vertical allocation can also be critically flawed, in the case when a device in charge of a whole layer loses communication with surrounding devices in the adjacent layers. In this context, the hybrid allocation in *EdgePipe* would outperform other allocations with the existence of volatile wireless links.

VI. Evaluation

We implemented *EdgePipe* using *TensorFlow* 1.15.0 in *Python* 3.7. We used the data sets of MNIST [48], Fashion-MNIST [49], EMNIST [50], and CIFAR-10 [51] with 60 000, 60 000, 60 000, and 50 000 training data, respectively, and 10 000 testing data to validate our proposed scheme. We primarily present the experimental results obtained using the Fashion-MNIST data set, unless otherwise noted. A DNN,

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SIMULATION ENVIRONMENT AND PARAMETERS

Neural Networks Settings	
# of training data	$50,000 \sim 60,000$
# of hidden layers	$2 \sim 8$ (mainly 4)
# of neurons in hidden layers	128
Activation function	ReLU
Optimizer	Gradient Descent
Learning rate	0.01
Batch size	100
Loss function	Softmax cross-entropy with digits
Wireless Edge Network Environment	
# of devices	$2 \sim 10$ (mainly 6)
Width and height of RoI	$25 \sim 100$ m
Path-loss exponent	$2.9 \sim 3.7$ (mainly 3.1)
Reference loss	46.68 dB
Additive white Gaussian noise	$N(0, 8^2)$ in dB

TABLE II
AVERAGED EXECUTION TIME FOR TRAINING WITH TEN TRIALS ON RASPBERRY PI 2 MODEL B IN MS (I: INPUT, H: HIDDEN, AND O: OUTPUT LAYER)

	Forward			*Sharing*	*Backward*	
Pass	I→H	H→H	H→O	O↔O	O→H	H→H
Vertical	341.42	49.04	4.24	1.85	6.94	79.34
Hybrid	161.15	24.87	2.19	1.48	2.30	21.27
Horizontal	56.09	8.66	1.49	1.51	0.87	14.86

TABLE III
EMPIRICAL TIMESLOT DURATION, THE NUMBER OF TIMESLOTS, TRAINING EPOCHS, BATCH STEPS, AND THE TOTAL NUMBER OF PROCESSED BATCH DATA FOR A DNN WITH SIX LAYERS USING SIX DEVICES DURING 194 MIN, WHEN THE TRAIN LOSS OF *Naive* BECOMES LOWER THAN 0.5

	Vertical	*Hybrid*	*Horizontal*	*Naive*
One timeslot time (ms)	541.77	311.33	172.86	441.42
# of timeslots	21510	37431	67416	26400
# of epochs, steps	18, 550	16, 356	11, 128	4, 600
# of trained batches	10750	9356	6128	2400

as mainly used in the experiments consists of one input layer and five dense layers with four hidden layers of 128 neurons, combined with a ReLU activation function and a gradient descent optimizer. The learning rate for the optimizer is 0.01, and a batch size of 100 is used. We used softmax cross-entropy with logits as the loss function. For efficient distributed learning, since the optimizers that need the intralayer communication may incur excessive costs, we used the basic optimizer.

For the main experimental environment with wireless edge devices, the simulated network consists of six edge devices which are randomly distributed over 50×50 m^2, as in shown Fig. 3(a). We used a combined path-loss shadowing model [52] with a path-loss exponent of 3.1, a reference loss of 46.68 dB, and an additive white Gaussian noise $\mathcal{N}(0, 8^2)$, resulting in an average PRR of 80.9 % between devices. The PRR between two devices is measured by sending 50 packets and counting the number of successfully transmitted packets within communication range. Although there are some other possible quality measures, we adopt the PRR, a simple and widely used link quality measure, instead of packet loss, which is directly related to network throughput [53]. The whole test environment is described in Table I. For matrix operations, we computed the output signal as if the input signal is received as 0 upon transmission failure in the forward passes. We ran 100 testing runs and calculated the average performance and the standard deviation. To run the neuron-to-device mapping algorithm, 1000 generations are executed until the Hadamard product is achieved in the steady state with the use of six devices. The larger the number of devices, the longer the network takes to saturate, due to the higher number of interconnected links. In the case of more than six devices, 10 000 generations are run.

We compared the performance results of *Hybrid* allocation in *EdgePipe* with three other counterpart allocation schemes: 1) *Vertical*: 1-Forward-1-Backward pipeline scheduling and weight stashing that takes the core mechanism from *PipeDream* [20]; 2) *Horizontal*; and 3) *Naive*: vanilla DNN running in a single device. Beyond the allocation, we investigated the scheduling efficiency using: 1) *Model*: a counterpart parallelism scheme based on model parallelism only, without pipeline acceleration and 2) *GPipe*: microbatch training from *GPipe* with the assumption of *Vertical* allocation for its expected full pipeline speedup by dividing a minibatch into the same number of microbatches as the number of devices [36], and another counterpart neuron-to-device mapping method, *Random*, which randomly maps the super neuron to the device.

To obtain the physical time range for the basic timeslot defined in pipeline scheduling, we measured the computation time taken to perform the computational tasks at a layer with its adjacent layer for the forward and the backward passes in a real-world edge device platform, Raspberry Pi Model B with 900-MHz quad-core ARM Cortex-A7 CPU and 1-GB RAM on Raspbian OS (Table II). For *GPipe*, the execution time is measured based on a *Vertical* allocation [20], by dividing by the number of microbatches. Since a fixed timeslot is required for pipelining scheduling, a timeslot is fixed to be the summation of the worst case computation time over all of the possible cases at a layer along the forward and backward passes, and the device-to-device transmission time with some time margin. For the communication time, the worst case transmission time with a low-power wireless interface of IEEE 802.15.4 is calculated as follows:

$$\left(M_{\text{pkt}} \times 32 \text{ bits/signal}\right) \div 250 \text{ kb/s} \tag{4}$$

where M_{pkt} is the maximum number of intermediate outputs; 784, 392, and 131 for *Vertical*, *Hybrid*, and *Horizontal*, respectively.

By taking into account the worst case latency in both computation and communication with a margin of 100 ms, a timeslot time for each allocation used for training was determined (Table III). For example, the worst computation time in *Hybrid* is 161.15 ms in the forward pass from an input layer to the first hidden layer, and the communication time

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

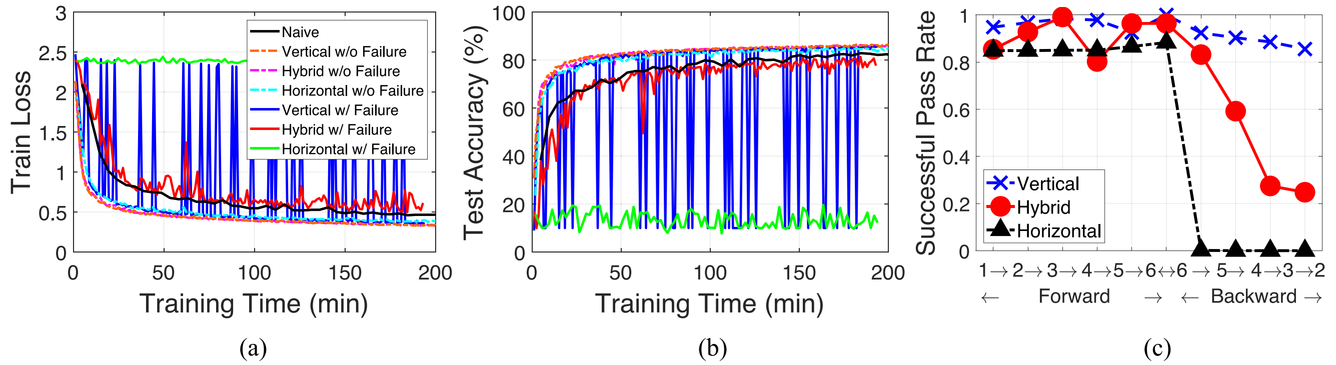


Fig. 5. Performance on training loss and test accuracy with respect to the training time, measured every 0.1 epochs. (a) Train loss with respect to time. (b) Test accuracy with respect to time. (c) Averaged successful layer-to-layer propagation rate.

is (392 × 32 bits/signal) ÷ 250 kb/s = 50.18 ms, resulting in $161.15 + 50.18 + 100 = 311.33$ ms in total. Although *Vertical* has a longer processing time, it can process more batches with the help of a pipeline. *Horizontal* distributes neurons from a single layer to all of the devices and, therefore, it cannot fully utilize acceleration. Our *Hybrid* has an advantage over both *Vertical* and *Horizontal*. Since *Naive* takes place in a single device, the transmission time is not counted. *GPipe* based on a *Vertical* allocation takes more communication overhead, due to the smaller size of transmitted microbatches instead of minibatches, with a factor of the number of splits. Based on this approach, the entire training time in the edge network is calculated by counting the number of timeslots required for each allocation type.

A. Distributed Learning

We validated *EdgePipe* in the aspects of model partitioning, neuron-to-device mapping, and pipeline scheduling. First, we investigated training loss and test accuracy in a network with two different link statuses: 1) without any link failure and 2) with link failures, against *Naive*, *Vertical* [20], and *Horizontal*, as in Fig. 5(a) and (b). In an ideal environment without any communication failures during a fixed training time, *EdgePipe (w/o Failure)* surpasses *Naive*, which does not have any failure either, by taking advantage of pipelining and making a model converge earlier, while all of the approaches show almost same training and testing performance. In a real-world environment, wireless link volatility badly affects the training and testing performance with major spikes in degradation over time. *Hybrid* in *EdgePipe* shows relatively stable training and testing performance with lower spikes than *Vertical*, reaching a higher testing accuracy than *Horizontal*, but falling slightly short of its own ideal version, *Hybrid* without failure. *Vertical* shows very unstable performance over time, due to the risk of complete loss of intermediate outputs, while *Horizontal* is almost untrained, leading to the worst test accuracy performance, since the training model failed to converge due to the frequent backpropagation failures.

We examined the way in which the layer-to-layer propagation path survived under link failures for each algorithm, as shown in Fig. 5(c). Overall, *Vertical* propagates the learning update from one layer to another over the forward and backward passes in the average sense. This is because its inherent device-to-device communication across adjacent layers is relatively low compared to those of the other approaches. However, if a device in charge of a whole layer becomes inactive in the worst case, the entire learning process is collapsed, showing the issue of instability. This observation implies that as the network becomes versatile, *Vertical* has a high risk of training degradation. However, when a DNN model is partitioned in a horizontal way, there are relatively more distinct device pairs in charge of the adjacent layers, resulting in excessive transmission costs and, thus, the gradient computation would likely fail upon any single transmission loss, blocking the entire training run. Furthermore, the horizontal allocation structure cannot utilize pipeline acceleration. This situation has the important implication that distributing the neurons across somewhat long consecutive layers to a single device should be prohibited in a wirelessly connected edge device network. In the case of *Hybrid*, although it sometimes shows unsuccessful layer-to-layer propagation performance, particularly at the backward passes, some effective neurons across adjacent layers for each device turn out to be resilient to the volatility, keeping up relatively stable training progress.

We varied the maximum number of layers in the super neuron, M_{layers}, in Fig. 6. To investigate the way in which each pipeline scheduling results in the reduction of training, the training time is quantified based on the number of timeslots spent, until the same amount of training process is reached at epoch 20, as shown in Fig. 6(a). *Vertical* takes full advantage of pipeline speed-up by dividing the pipeline stage into the finest level, having six pipeline stages from the input layer to the output layer, whereas *Horizontal* cannot benefit from pipelining at all. Although *Horizontal* consumes more timeslots until 20 epochs than other allocation schemes, the timeslot duration is shorter, resulting in a reasonable training time. Our *Hybrid* approach is positioned between these two extremes and shows performance competitive with that of $M_{\text{layers}} = 2$, while spending only 249 min for training with 3.75 packet transmissions per second on average at each Raspberry Pi-level device. We demonstrated that pipeline parallelism achieved considerable improvement, in test accuracy by reducing the training time with a factor of training speedup in both *Vertical* and *Hybrid* allocations.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

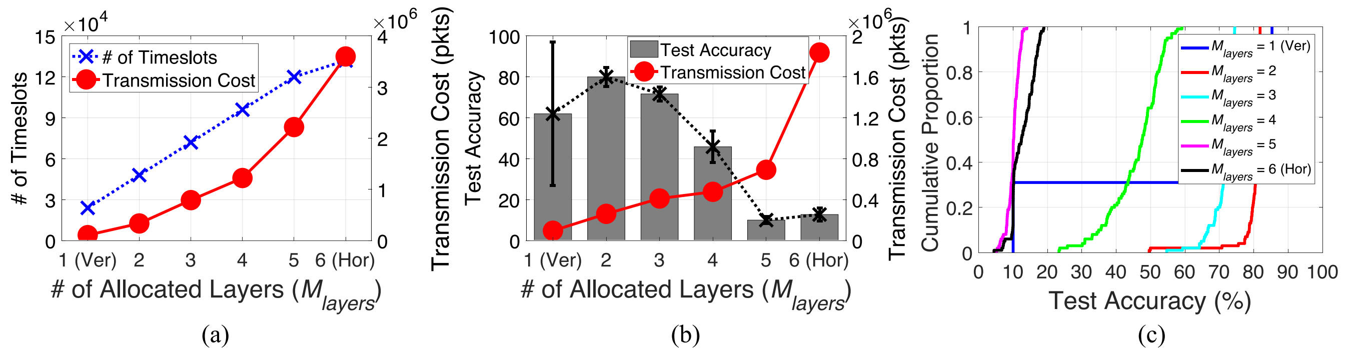


Fig. 6. Performance on training timeslots, communication overhead, and test accuracy with respect to M_{layers}. (a) Training overhead until 20 epochs. (b) Test accuracy and transmission cost with respect to M_{layers} at 194 min (c) Cumulative distribution with respect to test accuracy at 194 min over 100 test runs.

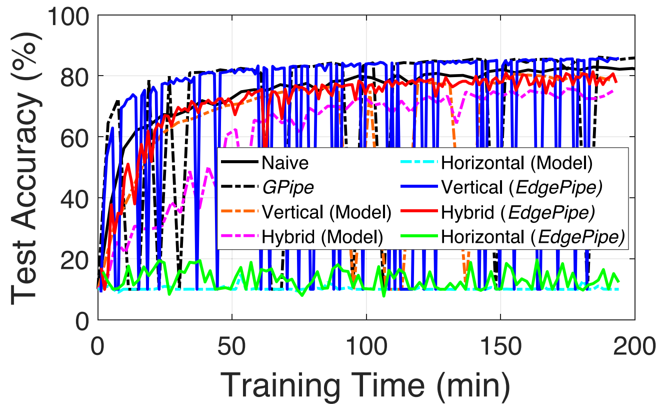


Fig. 7. Effect of pipeline acceleration in *EdgePipe*.

Regarding the distributed neuron structure for edge devices in Fig. 6(b), *Hybrid* with $M_{\text{layers}} = 2$ shows the best and most stable performance compared to *Vertical* and *Horizontal*, with a performance gap of 18.05 % and 67.25 %, respectively. We verified that our *Hybrid* allocation tries to invest at least two devices in charge of single layer from (1), not only taking advantage of pipeline acceleration but also showing resilience with a few spikes. If we take a closer look at the test accuracy distribution according to M_{layers}, as in Fig. 6(c), as the number of allocated layers, M_{layers}, increases, an allocation scheme is likely to have training failures on the backpropagation passes between adjacent layers. In the case of *Vertical*, if even a single failure occurs in one of the intermediate passes, the ongoing calculation to the subsequent layers totally collapses, showing a binary distribution. This implies that there exists a suitable selection of the maximum number of layers of which a device needs to take charge. A super-neuron-based network across adjacent layers contributes to making a distributed DNN relatively less fragile under uncertainty.

We validated the component of learning acceleration based on pipeline parallelism in *EdgePipe*, compared to model parallelism only without pipelining, and the microbatch-based parallelism of *GPipe*, as shown in Fig. 7. We show the dynamics of test accuracy with respect to the training time. The training time is converted from the number of epoch counts, as shown in Table III. We demonstrated that pipeline parallelism achieved great improvements in test accuracy by reducing the training time with a factor of 2.47 training speedup, compared to *Model Parallelism* to achieve the same inference performance (>80.0 %). *GPipe* partitions the model across the layers and, thus, shows similar performance to that of our *Vertical* allocation since the microbatch-based parallelism of *GPipe* needs to rely on *Vertical* allocation for its highest pipeline acceleration. For this reason, *GPipe* shows relatively unstable learning performance with sudden accuracy drops over time. This observation indicates that the way in which the model is partitioned is a key factor in volatile learning, and the pipeline approach contributes to accelerating learning.

Horizontal allocation falls behind compared to the other allocations, since the training model failed to converge, due to the frequent backpropagation failures. There also exists a tradeoff between pipeline speed-up benefit and real-world link volatility, *Vertical* used the highest pipeline efficiency, but becomes fragile under the volatility, while *Horizontal* cannot take advantage of the pipeline, but maintains stable training under volatility. *EdgePipe* with *Hybrid* allocation shows relatively more stable learning, compared to the microbatch-based parallelism of *GPipe*.

We implemented different neuron-to-device allocations to investigate the way in which a neuron-to-device mapping scheme affects training (Fig. 8). Our genetic algorithm for finding a mapping from a device-to-device network topology to a super-neuron-based network topology plays a key role in stable training taking only 2.86 s for six devices on a Raspberry Pi 2 Model B. It is compared against *Random*, which randomly maps between two topologies over five trials, and *Upper*, which identifies the mapping by trying out all possible cases in a brute-force manner, and is considered to be the upper bound. In the case of Vertical, our mapping algorithm generates the device deployment with *Upper*. As indicated in Fig. 8, our mapping scheme outperforms *Random* in all of the partitioning cases, and shows similar learning performance as *Upper*. Specifically, *Vertical* shows a dramatic performance gain, where the number of effective links is relatively small. This finding implies that it is important to extract the "skeletons" of wireless links. When all of the device-to-device connections are necessary, as in *Horizontal* allocation, the algorithm has similar training progress. By analyzing the experimental results from Figs. 6–8, we can conclude that both the super neuron structure and the super neuron-to-device

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

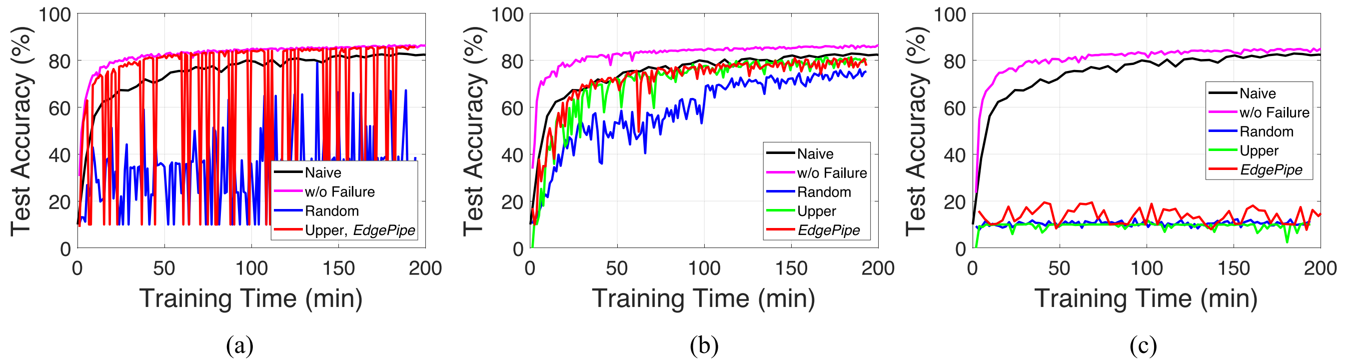


Fig. 8. Dynamics with respect to neuron-to-device allocation. (a) Vertical. (b) Hybrid. (c) Horizontal.

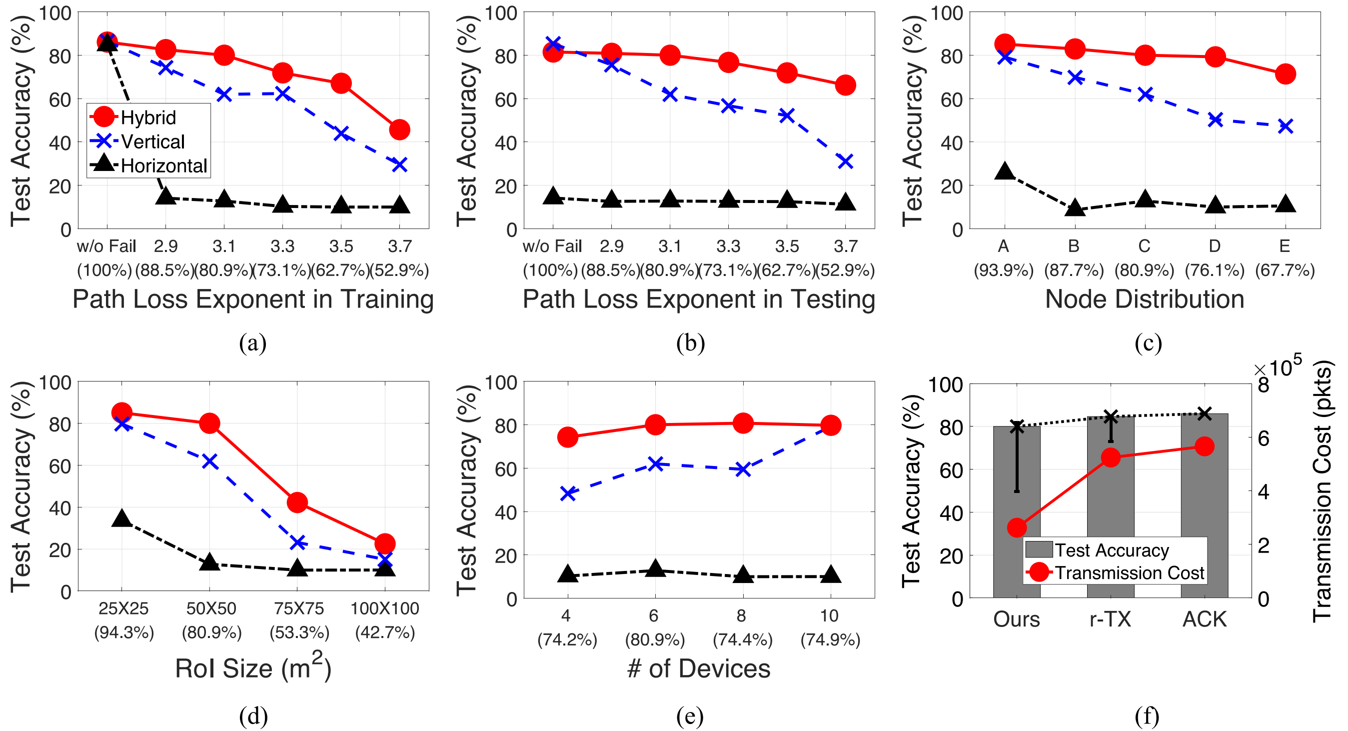


Fig. 9. Effects of various network configurations (where the average PRR (%) is shown in parentheses) and failure recovery schemes. (a) Degree of link volatility in training. (b) Degree of link volatility in testing. (c) Different device distribution in training. (d) Network RoI size in training. (e) # of edge devices in training. (f) TX recovery scheme in training.

mapping with pipeline acceleration are essential components for designing edge DNNs, in order to achieve high stable performance with fast training time, which is boosted by pipelining.

B. Resilience Under Various Learning Environments

To inspect the resilience under various learning environments, we tested *EdgePipe* with different various network configurations (Fig. 9). First, we verified how *EdgePipe* can endure different degrees of link volatility by varying the path-loss exponent to form different network PRRs over the same device-to-device topology, with a specific path-loss exponent used for both training and testing [Fig. 9(a)], and with various path-loss exponents only at testing with the path-loss exponent of 3.1 used at training [Fig. 9(b)]. *Hybrid* retains the highest performance in most cases, and is not seriously affected under even worse network situations, whereas *Vertical* is severely damaged by network conditions.

We investigated the test accuracy under different device distributions [Fig. 9(c)]. Even though the same edge resource is deployed in the same territory area, the node distribution produces a high variation in network connectivity; *Vertical* is markedly affected, with a 28.8 % performance gap. However, *Hybrid* allocation maintains relatively stable inference, with a 13.76 % performance gap between the best and the worst cases, across different node distributions, outperforming *Vertical* by a factor of 1.58 at most. The edge-side learning is highly dependent on the wireless volatility, and constructing a stable learning architecture in a lossy environment is essential. The experimental results shown in Fig. 9 have demonstrated that the device topology does affect the PRR, causing severe deterioration in a distributed learning architecture with *Vertical* allocation.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

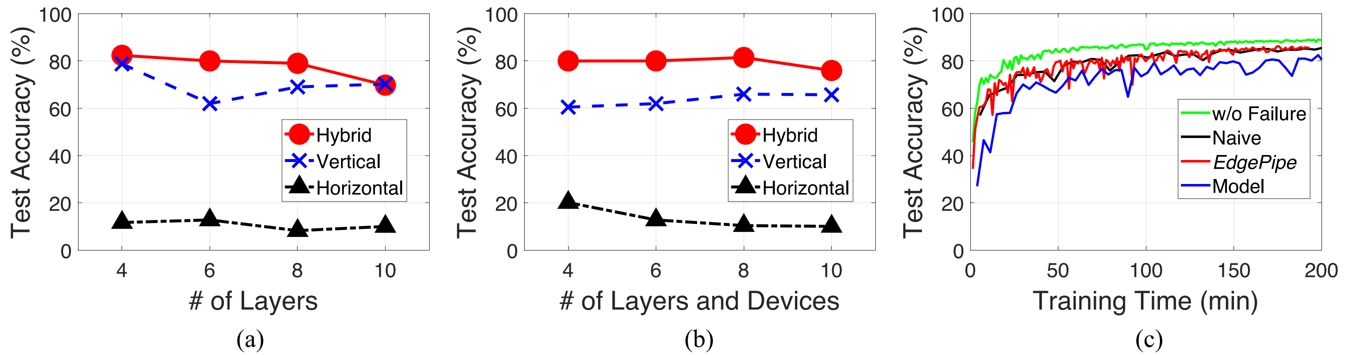


Fig. 10. Effects of different structures in a network model. (a) Number of layers in a DNN. (b) Number of layers and devices in a DNN. (c) Training model using CNN.

As the size of network Region of Interest (RoI) increases, as in Fig. 9(d), or the number of devices decreases, as in Fig. 9(e), the devices are located more sparsely, and packet transmission among them becomes less successful. Therefore, in the case of *Hybrid* allocation, the inference performance becomes degraded due to training failure caused by link failure, whereas *Vertical* has an even lower performance due to more frequent failures at the intermediate passes in testing. This situation means that *Hybrid* with more training could alleviate the degradation, even under poor connectivity, while *Vertical* becomes inferior due to its innate inference breakdown. In the same context, although *Vertical* becomes as reliable as *Hybrid* given sufficient resources with stable connectivity for learning and inference, the fluctuating links can cause severe deterioration in a distributed learning architecture with *Vertical* allocation. This finding implies that edge-side learning is highly dependent on wireless volatility, and constructing a stable learning architecture in a lossy environment is essential.

For a given RoI, we investigated how adding device resources affects test accuracy, by varying the number of edge devices, as shown in Fig. 9(e). *Hybrid* performs well, and better than *Vertical* and *Horizontal* when some minimum number of devices are available, with a performance gap of 25.98 % and 64.02 %, respectively, using four devices. When a sufficient number of devices is used, *Vertical* becomes similar to *Hybrid*, sharing the neurons from a single layer with other devices. *Vertical* becomes as reliable as *Hybrid* with sufficient device resources for learning and inference.

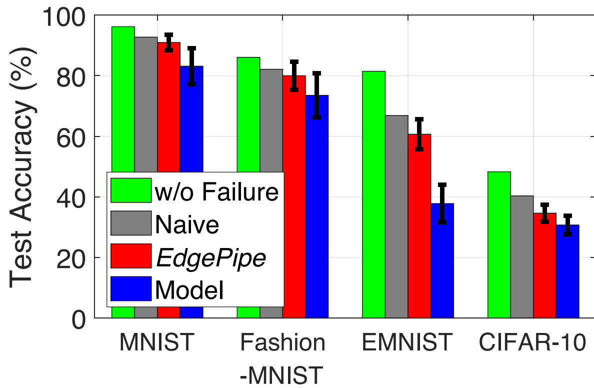
To investigate the way in which different transmission recoveries can improve performance with additional packet overhead, we used some transmission (TX) recovery schemes: 1) *r-TX*: a simple redundant transmission scheme with two consecutive times and 2) *ACK*: a multiple retransmission scheme until acknowledgement is successfully received. Those link recovery schemes are supposed to show the same accuracy performance in an ideal failure-free environment. The original *EdgePipe* without any link recovery called *Ours* is compared with those with *r-TX* and *ACK*, respectively, for each different allocation, as in Fig. 9(f). The link failure has been compensated for increasing the test accuracy from 80.01 % to 84.62 %, and to 85.97 %, while investing more transmission with a factor of 2.0 and 2.16, respectively. However, the proposed *Hybrid* allocation of *EdgePipe* performs well even under network volatility, without the help of additional link recovery. This result implies that forming a resilient neuron group structure is the key to designing a reliable yet efficient distributed learning framework despite network volatility. *EdgePipe* offers a concrete way to provide such a benefit, thanks to the *super neuron* structure.

C. Feasibility and Scalability of EdgePipe

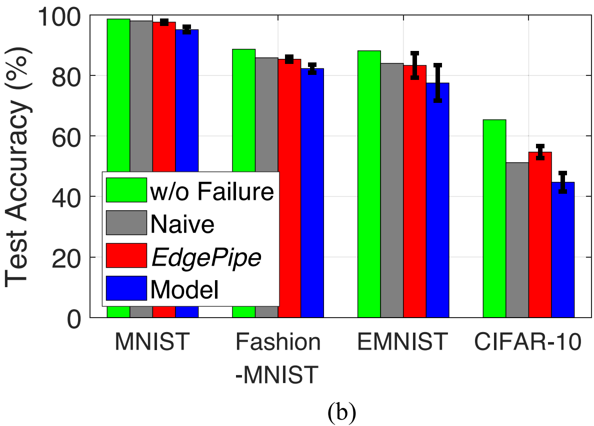
We examined in-depth resilience performance under different learning environments from the perspectives of different neural network models (Fig. 10). By varying the number of layers in a DNN structure given the same number of devices, as shown in Fig. 10(a), both *Hybrid* and *Vertical* maintain high performance using the same number of devices with link failures. When the number of layers becomes deeper than six, using six devices, a single device mostly takes charge of a whole layer in *Vertical*, and the model would probably collapse, even with a single link failure between adjacent layers. With the limited resources, as the neural networks become larger, *Hybrid* leans toward *Horizontal* to prevent the inference failure from allocating the whole layer to a single device. At the same time, as *Hybrid* moves relatively away from *Vertical* and loses its pipeline acceleration to some degree, *Hybrid* ends up with slower training performance than *Vertical*, but finally achieves higher test accuracy performance, in most cases. We validated the consistency of our approach using the same amount of resources by investing the same number of devices as the number of layers. As illustrated in Fig. 10(b), *Hybrid* outperforms the other allocation schemes under fair resource conditions.

We also investigated our scheme based on a different network model, which is CNN [Fig. 10(c)]. We used three convolution layers and four fully connected layers, where the number of filters in the first convolution layer is 32, and the number in the other layers are 64 with a size of 3×3. The max-pooling layers with a size of 2×2 are stacked after the second and third convolution layers. A stride of 2 is used. CNN achieved better performance than DNN, with a performance improvement of 2.52 % in *w/o Failure*, 4.32 % in *EdgePipe*, and 5.70 % in *Model* (which is an *EdgePipe* version without pipeline acceleration). Since CNN allows the achievement of

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

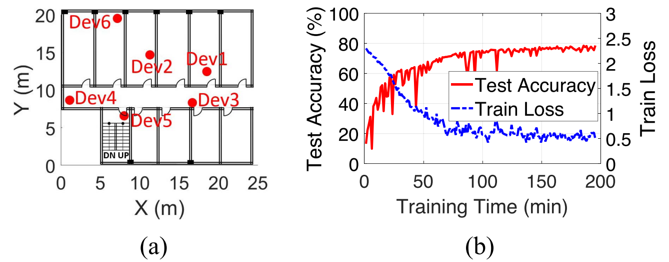


(a)



(b)

Fig. 11. Effect of different data sets with (a) DNN and (b) CNN, respectively.



(a) (b)

Fig. 12. *EdgePipe* in a real-world wireless testbed. (a) Real-world device deployment testbed. (b) Dynamics of train loss and test accuracy.

stable high performance much faster than DNN (with 50 min training, the accuracy reaches at 80.14 % with CNN and 67.23 % with DNN), *Model* that does not utilize pipeline acceleration performs well, while *EdgePipe* outperforms the one without pipeline acceleration.

We validated *EdgePipe* over various data sets using both DNN and CNN (Fig. 11), in terms of accuracy. As in both Fig. 11(a) and (b), although *EdgePipe* performs a little worse than its ideal version without any communication failures due to the innate volatility, it still shows very competitive performance better than the model parallelism-only approach. Assuming an ideal failure-free environment, *EdgePipe* denoted as *w/o Failure* shows accuracy similar to that of *Naive* which is free from any communication issue, with only 194 min of training, thanks to pipeline acceleration. Furthermore, when CNN is employed instead of DNN, as shown in Fig. 11(b), our *EdgePipe* scheme performs well on various data sets, with some convolution and pooling layers with CNN, achieving up to 97.64 %, 85.38 %, 83.33 %, and 54.68 % with respect to the data set under the fragile network topology, while matching or even outperforming, the performance of *Naive* via the training speedup. Due to the inherent strength of the convolutional layers, CNN achieves better inference than DNN. However, it should be noted that CNN is one type of DNN, which also includes dense layers. It implies that elaborate partitioning of the computationally intensive dense layer is essential in model parallelism, and additional layers, such as convolutional layers or pooling layers can be stacked, if necessary.

Finally, we evaluated *EdgePipe* based on real-world wireless measurements at a 20×20 m^2 testbed inside a university building using six TinyOS-based TelosB motes using the IEEE 802.15.4 radio (Fig. 12). The average PRR during the initial 1000 transmissions reached 71.62 %, and was used for neuron-to-device mapping. We verified that our *Hybrid* allocation with pipelining still offers feasible learning and inference, with the performance up to 78.98 % during 194 min, even under real-world wireless networks.

VII. Discussion

The experimental findings in this work raise several interesting follow-up discussions.

A. On-Device Learning With Link Volatility

As the intelligence moves to the end devices, on-device learning at resource-constrained and wireless edges makes the problem itself even more complicated [19], [54]. As demonstrated in Fig. 9, although the same amount of resources is used in federated learning, both training and inference are heavily affected, due to the lossy connection. Since the volatile wireless characteristics degrade distributed learning, the well-known existing distributed learning approaches [9], [20] cannot be directly applied. Wireless communication overhead is vital to the power consumption, making edge-side learning more challenging. To cope with the uncertainty in device-to-device communication, model partitioning based on neurons across some consecutive layers to each device, along with pipeline acceleration, turns out to be effective, maintaining high prediction performance.

B. Implications

1) Vertical Versus Horizontal: As a DNN model is partitioned in a *Horizontal* way, there are relatively more distinct device pairs in charge of the adjacent layers and, thus, the gradient computation would fail with high probability upon any single transmission loss. As explicitly shown in Fig. 6, the severe performance degradation in *Horizontal*, which has very low performance even for the best case, comes from the fact that the overall training has completely failed. The horizontal allocation structure cannot utilize pipeline acceleration, either. We can derive an important implication that distributing the neurons across somewhat long consecutive layers to a

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

single device should be prohibited in a wirelessly connected edge device network.

When a DNN model is instead partitioned in a *Vertical* way, a distributed deep learning structure can fully utilize pipeline acceleration, achieving shorter training times and higher accuracy compared to the others, by compensating for the link failures. However, the *Vertical* structure turns out to be very fragile, since a transmission loss from a single device usually in charge of a whole layer, breaks down all of the ongoing computation across the remaining preceding layers. As shown in Fig. 6, since this structure has shown very low worst case performance, the vertical allocation becomes more vulnerable at the inference phase.

2) Desirable Partitioning: Based on the implication between vertical and horizontal partitioning, a suitable mixture between vertical and horizontal in a hybrid way offers relatively more stable learning and inference performance in volatile wireless networks. As indicated in Fig. 2, desirable model partitioning needs to be positioned toward *Vertical* to obtain layer-to-layer propagation, cost reduction, and acceleration benefit from pipelining, but still preventing a single device from being involved with the whole layer. Thus, *Hybrid* allocation is derived as in (1), to involve at least two devices with a single layer, while making the model as vertical as possible.

3) Neural Network-to-Device Allocation: As the device-to-device connectivity varies within a wireless network, distributing partitioned networks to a suitable device is a very important step, as shown in Fig. 3. As the forward and backward passes should be executed in order, it is important to allocate a strongly connected pair of devices into the adjacent layers. Thus, the neural network-to-device allocation procedure should be derived considering not only neuron groups across some adjacent layers to a device but also the device-to-device connectivity in a network topology, for designing high-quality distributed learning.

VIII. Conclusion

We have presented *EdgePipe*, a novel pipelined model parallelism framework resilient to volatile edge networks. The proposed scheme introduced a versatile allocation structure, the *super neuron*, by partitioning the model at the level of both layers and neurons. We exploited a pipeline paradigm into model parallelism to compensate for delayed learning due to the wireless failures. We have demonstrated that *EdgePipe* constructs a reliable deep learning model and achieves stable inference under a lossy environment.

In future work, we may consider the staleness of the parameters in pipelining, to accelerate the learning process. Since the model suffers from consecutive backpropagation failure, it would be interesting to design a semioptimizer that can process a local update based on partial information. Extensive work that attempts to validate the idea using a range of neural network architectures, such as convolution networks or recurrent networks, will be an important future research direction. To tackle the inherently fluctuating connectivity among the devices more actively, some post-adjustment effort could alleviate network volatility variation in future work.

References

[1] R. Mayer and H.-A. Jacobsen, “Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools,” *ACM Comput. Surveys*, vol. 53, no. 1, pp. 1–37, 2020.
[2] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” 2018, *arXiv:1807.05358*.
[3] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[4] M. Li *et al.*, “Scaling distributed machine learning with the parameter server,” in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement.*, 2014, pp. 583–598.
[5] D. Rothchild *et al.*, “FetchSGD: Communication-efficient federated learning with sketching,” 2020, *arXiv:2007.07682*.
[6] S. Wang, D. Li, and J. Geng, “Geryon: Accelerating distributed CNN training by network-level flow scheduling,” in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2020, pp. 1678–1687.
[7] A. Xu, Z. Huo, and H. Huang, “On the acceleration of deep learning model parallelism with staleness,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 2088–2097.
[8] R. J. Hewett and T. J. Grady, II, “A linear algebraic approach to model parallelism in deep learning,” 2020, *arXiv:2006.03108*.
[9] L. Song, X. Qian, H. Li, and Y. Chen, “Pipelayer: A pipelined reram-based accelerator for deep learning,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 541–552.
[10] J. Zhan and J. Zhang, “Pipe-Torch: Pipeline-based distributed deep learning in a gpu cluster with heterogeneous networking,” in *Proc. 7th Int. Conf. Adv. Cloud Big Data (CBD)*, 2019, pp. 55–60.
[11] L. Guan, W. Yin, D. Li, and X. Lu, “XPipe: Efficient pipeline model parallelism for multi-GPU DNN training,” 2019, *arXiv:1911.04610*.
[12] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, “Pipe-SGD: A decentralized pipelined sgd framework for distributed deep net training,” in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Assoc., 2018, pp. 8045–8056.
[13] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, “FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 553–564.
[14] J. Ren, G. Yu, and G. Ding, “Accelerating dnn training in wireless federated edge learning system,” 2019, *arXiv:1905.09712*.
[15] H. Li, K. Ota, and M. Dong, “Learning IoT in edge: Deep learning for the Internet of Things with edge computing,” *IEEE Netw.*, vol. 32, no. 1, pp. 96–101, Jan./Feb. 2018.
[16] J. Chen and X. Ran, “Deep learning with edge computing: A review,” *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.
[17] H. Hu, D. Wang, and C. Wu, “Distributed machine learning through heterogeneous edge systems,” in *Proc. AAAI*, 2020, pp. 7179–7186.
[18] J. Park, S. Samarakoon, M. Bennis, and M. Debbah, “Wireless network intelligence at the edge,” *Proc. IEEE*, vol. 107, no. 11, pp. 2204–2239, Nov. 2019.
[19] G. Zhu, D. Liu, Y. Du, C. You, J. Zhang, and K. Huang, “Toward an intelligent edge: Wireless communication meets machine learning,” *IEEE Commun. Mag.*, vol. 58, no. 1, pp. 19–25, Jan. 2020.
[20] A. Harlap *et al.*, “PipeDream: Fast and efficient pipeline parallel DNN training,” 2018, *arXiv:1806.03377*.
[21] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, “MALT: Distributed data-parallelism for existing ML applications,” in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–16.
[22] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, “Enhancing data parallelism for ant colony optimization on GPUs,” *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, 2013.
[23] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, “Measuring the effects of data parallelism on neural network training,” 2018, *arXiv:1811.03600*.
[24] J. Subhlok, J. M. Stichnoth, D. R. O’hallaron, and T. Gross, “Exploiting task and data parallelism on a multicomputer,” in *Proc. 4th ACM SIGPLAN Symp. Principles Pract. Parallel Program.*, 1993, pp. 13–22.
[25] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: Using data parallelism to program gpus for general-purpose uses,” *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 325–335, 2006.
[26] C. Hardy, E. Le Merrer, and B. Sericola, “Distributed deep learning on edge-devices: Feasibility via adaptive compression,” in *Proc. IEEE 16th Int. Symp. Netw. Comput. Appl. (NCA)*, 2017, pp. 1–8.
[27] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, “A hierarchical model for device placement,” in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–11.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.

[28] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-LM: Training multi-billion parameter language models using GPU model parallelism,” 2019, *arXiv:1909.08053*.
[29] A. Mirhoseini *et al.*, “Device placement optimization with reinforcement learning,” 2017, *arXiv:1706.04972*.
[30] R. Mayer, C. Mayer, and L. Laich, “The tensorflow partitioning and scheduling problem: It’s the critical path!” in *Proc. 1st Workshop Distrib. Infrastruct. Deep Learn.*, 2017, pp. 1–6.
[31] D. Das *et al.*, “Distributed deep learning using synchronous stochastic gradient descent,” 2016, *arXiv:1602.06709*.
[32] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “Hypar: Towards hybrid parallelism for deep learning accelerator array,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2019, pp. 56–68.
[33] S. Pal *et al.*, “Optimizing multi-GPU parallelization strategies for deep learning training,” *IEEE Micro*, vol. 39, no. 5, pp. 91–101, Sep./Oct. 2019.
[34] J. Dean *et al.*, “Large scale distributed deep networks,” in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Assoc., 2012, pp. 1223–1231.
[35] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, “Efficient and robust parallel DNN training through model parallelism on multi-GPU platform,” 2018, *arXiv:1809.02839*.
[36] Y. Huang *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Assoc., 2019, pp. 103–112.
[37] J. H. Park *et al.*, “HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism,” 2020, *arXiv:2005.14038*.
[38] L. Xiao, X. Lu, T. Xu, X. Wan, W. Ji, and Y. Zhang, “Reinforcement learning-based mobile offloading for edge computing against jamming and interference,” *IEEE Trans. Commun.*, vol. 68, no. 10, pp. 6114–6126, Oct. 2020.
[39] S. Yu, X. Chen, L. Yang, D. Wu, M. Bennis, and J. Zhang, “Intelligent edge: Leveraging deep imitation learning for mobile edge computation offloading,” *IEEE Wireless Commun.*, vol. 27, no. 1, pp. 92–99, Feb. 2020.
[40] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, “Mobile edge computing: A survey,” *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.
[41] X. Tang, X. Chen, L. Zeng, S. Yu, and L. Chen, “Joint multi-user DNN partitioning and computational resource allocation for collaborative edge intelligence,” *IEEE Internet Things J.*, vol. 8, no. 12, pp. 9511–9522, Jun. 2021.
[42] M. M. Amiri and D. Gündüz, “Machine learning at the wireless edge: Distributed stochastic gradient descent over-the-air,” *IEEE Trans. Signal Process.*, vol. 68, pp. 2155–2169, Mar. 2020.
[43] M. M. Amiri and D. Gündüz, “Over-the-air machine learning at the wireless edge,” in *Proc. IEEE 20th Int. Workshop Signal Process. Adv. Wireless Commun. (SPAWC)*, 2019, pp. 1–5.
[44] A. Bavelas, “Communication patterns in task-oriented groups,” *J. Acoust. Soc. America*, vol. 22, no. 6, pp. 725–730, 1950.
[45] Y. Wu *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” 2016, *arXiv:1609.08144*.
[46] Y. Kang *et al.*, “Neurosurgeon: Collaborative intelligence between the cloud and mobile edge,” *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.
[47] R. A. Horn, “The hadamard product,” in *Proc. Symp. Appl. Math.*, vol. 40, 1990, pp. 87–169.
[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[49] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms,” 2017, *arXiv:1708.07747*.
[50] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, “EMNIST: Extending mnist to handwritten letters,” in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2017, pp. 2921–2926.
[51] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Rep. TR-2009, 2009.
[52] T. S. Rappaport *et al.*, *Wireless Communications: Principles and Practice*, vol. 2. Hoboken, NJ, USA: Prentice Hall PTR, 1996.
[53] D. C. Salyers, A. D. Striegel, and C. Poellabauer, “Wireless reliability: Rethinking 802.11 packet loss,” in *Proc. Int. Symp. World Wireless Mobile Multimedia Netw.*, 2008, pp. 1–4.
[54] Y. Huang, X. Ma, X. Fan, J. Liu, and W. Gong, “When deep learning meets edge computing,” in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, 2017, pp. 1–2.

JinYi Yoon received the B.S. and M.S. degrees in computer science and engineering from Ewha Womans University, Seoul, South Korea, in 2017 and 2019, respectively, where she is currently pursuing the Ph.D. degree.

Her current research interests include edge computing, future wireless networks, AI-driven network system design, and localization.

Yeongsin Byeon received the B.S. degree in computer science and engineering from Ewha Womans University, Seoul, South Korea, in 2021.

Her current research interests include future wireless networks on IoT, distributed deep learning, fog computing and mobility, and stream data.

Jeewoon Kim is currently pursuing the B.S. degree in computer science and engineering with Ewha Womans University, Seoul, South Korea.

Her current research interests include edge computing, wireless networks on the IoT, AI-driven network design, and user localization.

HyungJune Lee (Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2001, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2006 and 2010, respectively.

He joined Broadcom, San Jose, CA, USA, as a Senior Staff Scientist for working on research and development of 60GHz 802.11ad SoC MAC. He also worked for AT&T Labs, Atlanta, GA, USA, as a Principal Member of Technical Staff with the involvement of LTE overload estimation, LTE-WiFi interworking, and heterogeneous networks. He is currently an Associate Professor with the Computer Science and Engineering Department, Ewha Womans University, Seoul. His current research interests include future wireless networks on the IoT, fog computing, VANET, and machine learning-driven network system design.

Authorized licensed use limited to: Ewha Womans Univ. Downloaded on November 05,2022 at 00:36:53 UTC from IEEE Xplore. Restrictions apply.