Non-Linear Erasure Code for Distributed Inference: A Machine Learning-based Coding Approach

Anonymous

***Abstract*—Distributed inference enables multiple distributed computing devices to cooperatively perform statistical inference or data analysis tasks, reducing the inference delay. However, due to device heterogeneity, some of the distributed computing devices may have slow computing speeds or even fail in computing, which results in an increased inference delay or even inference failure. In this work, we propose a coded inference framework by introducing redundant computing devices. To achieve this framework, we design an encoder to determine the inference inputs of the redundant devices and a decoder to reconstruct the inference result. Designing the encoder and decoder is challenging due to the nonlinear computation involved in the inference process. To address this challenge, we propose a machine learning-based coding approach by introducing a neural network to the process of erasure-code computation. Our proposed approach is resilient to complex nonlinear computational contexts. Our experimental results on the image classification task demonstrate that accuracy does not significantly decrease even when certain distributed devices fail, showcasing the feasibility and advantages of this approach. (Comment: The numerical performance and insights.)**

***Index Terms*—Distributed Inference, Coded Computation, Deep Neural Networks**

I. Introduction

CONVOLUTIONAL neural networks (CNNs) have been implemented in various applications, such as image classification [1] and object detection [2]. Since CNN inference usually requires large computing resources, distributed inference [3], [4] has been introduced by exploiting distributed computation to CNN inference. Specifically, distributed inference enables multiple distributed computing devices to collaboratively execute an inference task. Through efficient inference task distribution and cooperation, distributed inference can accelerate the inference speeds and allow the handling of inference tasks over large datasets while reducing the load on individual computing devices. Many existing works (e.g., [5], [6]) have proposed layer-wise partitioning, which enables the parallelization of forward pass and backward pass. Xue *et al.* in [7] adopted layer-fusion optimization to reduce the communication overhead of frequent intermediate data exchange between devices.

Despite the success of the aforementioned works (e.g., [5]–[7]), in distributed inference systems, (i) devices may sometimes fail in determining the inference outputs (e.g., fail in computing, disconnected from the system due to its mobility), and (ii) different devices may have different computing speeds and bandwidth, which leads to straggler issue (i.e., the computing time of some devices is much longer than those of the others). The device failure can lead to the inference task failure. Meanwhile, the presence of the stragglers can severely increase the inference delay.

To address the challenges of device failure and straggler issues, in this work, we propose a coded inference framework by introducing device redundancy into the system. In this framework, we focus on one inference task. The original inputs of the inference task are encoded to generate redundant inputs. Both the original inputs and redundant inputs are assigned to a set of computing devices for performing inference processes. Given a carefully-designed encoding and decoding method, the result of the inference task can be determined (or equivalently, decoded) when a subset of the computing devices have accomplished their inference processes. The completion of the inference task depends on only whether the number of devices completing the inference is larger than a pre-defined value or not. Meanwhile, the inference delay depends on the delay of the devices which complete inference the earliest. Thus, such a coded inference framework can alleviate the device failure and straggler issues and hence reduces the inference delay.

The implementation of our proposed coded inference framework relies on the design of encoder and decoder. This is closed related to the existing works on coded computation. For example, Lee *et al.* in [8] proposed coded computation for matrix multiplication. Yu *et al.* in [9] use Lagrange polynomial to encode polynomial computation. Soleymani *et al.* in [10] research on binary linear codes and decode Reed-Muller codes. However, these aforementioned works [8]–[10] focused on linear computation and cannot support nonlinear computation. Note that *CNN inference usually involves non-linear computation due to the operation at the activation layer*, under which the aforementioned works are not applicable.

Some existing works have proposed nonlinear coded computation methods. Mallick *et al.* in [11] proposed to decompose a complex nonlinear computation into multiple simple nonlinear computation. However, this approach is unrealistic due to the requirement of model split. In [12], Prakash *et al.* used federated learning to enable data sharing between distributed devices to achieve redundant computing. However, data sharing increases the communication overhead, which is not an appropriate idea for distributed inference.

Designing the encoding and decoding methods in coded distributed inference is challenging due to the nonlinear activation layer of CNN. In this work, we propose a non-linear erasure code approach based on machine learning. Machine learning [13] is a powerful tool for addressing complex tasks, and hence it has the potential to overcome the non-linear challenges in distributed inference. The main idea is to implement neural networks as the encoder and decoder. Considering the non-linear operation between the encoder and decoder, the encoder should characterize sufficient input information, and the decoder should be able to reconstruct the correct inference result based on the inference outputs of the encoded inputs. Note that such an encoder and decoder idea is inspired by [13]. Kosaian *et al.* in [13] proposed to use machine learning to design an encoder and decoder, providing a feasible solution for applying coded computation to a nonlinear model and obtaining approximate results. Different from [13], in our work, we adopt the machine learning-based encoding idea into distributed inference. We propose the method to split and encode inputs for distributed inference and design specific encoder and decoder structures for inference models.

Our main contributions are summarized as follows:

- **Coded Inference Framework:** We introduce a novel coded distributed inference framework that allows distributed computing devices to cooperatively perform an inference task while being resilient to device failure and straggler issues. This framework
- **Encoder and Decoder Design:** To address the non-linear function, we propose two pairs of encoder and decoder architectures based on multi-layer perception (MLP) and CNN respectively. The MLP-based architecture is suitable to the scenario where computational resources are limited, or real-time inference is required. Its fully connected layers effectively captures complex relationships among input features. This architecture excels in many problems, particularly in tasks involving sequential data or images. On the other hand, the CNN-based architecture better handles spatial features, such as images or two-dimensional data. CNN architectures leverage convolutional and pooling layers to share weights across different regions of the input data, enabling them to capture local features and structural information more effectively. The shared weight characteristic helps mitigate overfitting, and as a result, the CNN-based architecture achieves higher accuracy in inference results.
- **Performance Evaluation:** We evaluate the performance of our proposed coded distributed inference framework using MNIST, CIFAR-10, and CIFAR-100 datasets. The outcomes of experiments consistently indicate that our framework maintains a high level of base models' accuracy, affirming its robustness and effectiveness.

The rest of this paper is organized as follows. Section II details our coded inference design. The encoder and decoder architectures are elaborated in Section III. Section IV presents the performance evaluation. Section V concludes this work.

II. Coded Inference Framework

In our framework, we incorporate encoding and decoding techniques into distributed inference and enable redundant devices to participate in the inference process. In the occurrence of device failure or stragglers, this framework can leverage the available outputs (i.e., the outputs of the devices that have completed their inference processes) to reconstruct the inference result. Thus, this framework can enhance the robustness of the distributed inference system.

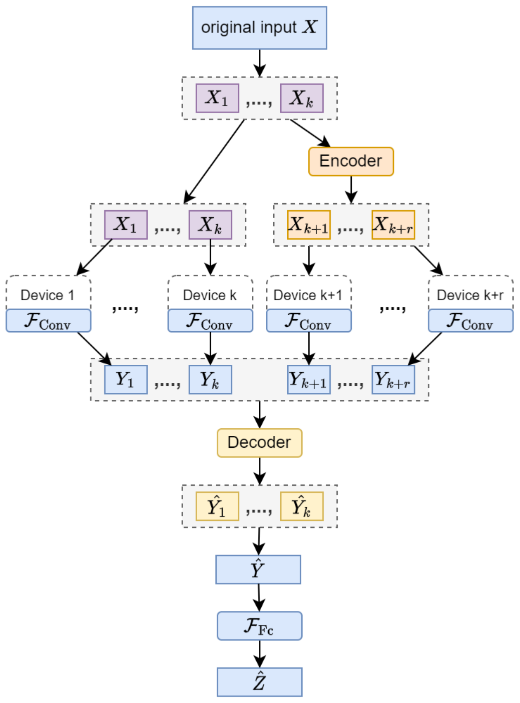


Fig. 1: The illustration of coded distributed inference framework. Components related to encoding and decoding are colored with orange and yellow, respectively. Redundancy is introduced, allowing for inference result recovery even if the inference processes of up to r computing devices fail.

A. Framework Details

Consider a coded distributed inference system with a total of n computing devices. We designate k devices for processing the original input data, and designate r devices for handling the redundant inputs generated by an encoder. Thus, $k+r=n$. The neural network used for inference is denoted by $\mathcal{F}$. This nerual network $\mathcal{F}$ is separated into two distinct segments: a convolutional segment $\mathcal{F}_{Conv}$ and a fully-connected segment $\mathcal{F}_{Fc}$. As in many existing works on distributed inference (e.g., [??]), we consider the setting where $\mathcal{F}_{Conv}$ is performed on n distributed devices, and $\mathcal{F}_{Fc}$ is performed by an arbitrary device in a centralized fashion. This is because a distributed inference operated over $\mathcal{F}_{Fc}$ will lead to significant communication overhead (due to the fully connected nature) while a limited inference delay reduction (due to the low computational complexity of $\mathcal{F}_{Fc}$ when compared with $\mathcal{F}_{Conv}$).

Figure 1 illustrates our proposed coded distributed inference framework. We consider an inference task with input X, e.g., an image in an image classification task. An encoder $\mathcal{E}$ and a decoder $\mathcal{D}$ are introduced into the distributed inference system.

Input splitting: The original input X is first split into k distinct parts $X_1, \dots, X_k$, e.g., in an image classification task, an image is split into k image pieces. By splitting the input X into $X_1, \dots, X_k$, the objective is to divide the inference outputs of the computing devices (over inputs $X_1, \dots, X_k$) into equal sizes, with which the outputs can be fed into the decoder. Due to the inter-dependency between layers in

$\mathcal{F}_{Conv}$, each input part X_i partially overlap with its adjacent segments X_{i-1} and X_{i+1}. The overlapping range is computed based on the specific CNN structure in order to ensure the required shape of the outputs and to avoid the computing devices from sharing their intermediate results during their independent inference processes.

Encoding process: The k split inputs are encoded to compute r redundant inputs, $X_{k+1}, ..., X_{k+r}$. These r inputs, along with the original k inputs, form the $n = k + r$ inputs for distributed inference, denoted as $X_1, X_2, ..., X_k, ..., X_n$.

Distributed inference: The n inputs, $X_1, X_2, ..., X_n$, are assigned to n computing devices. Each computing device $i = 1, \ldots, n$ performs inference independently with an input X_i using a copy of the non-linear convolutional segment $\mathcal{F}_{Conv}$ and obtains an output Y_i. The output is then forwarded to a pre-chosen computing device for decoding.

Decoding process: When any arbitrary subset of k computing devices (out of n computing devices) have accomplished their inference processes, their inference outputs are fed into the decoder $\mathcal{D}$, resulting in k predicted convolutional outputs $\hat{Y}_1, \hat{Y}_2, \ldots, \hat{Y}_k$. Note that $\hat{Y}_i$ does not necessarily correspond to the decoded output of X_i, while they correspond to the decoded output of the k fastest computing devices. Then, these outputs are amalgamated into a single output $\hat{Y}$ by xxx, where $\hat{Y}$ corresponds to an approximate inference result $\hat{Y} = \widehat{\mathcal{F}}_{Conv}(X)$ of the convolutional segment $\mathcal{F}_{Conv}$.

Inference at Fully connected layer: The output $\hat{Y}$ is then fed to the fully-connected segment $\mathcal{F}_{Fc}$, mapping the enriched feature set into a final output space. This process of computation is crucial for integrating the high-level features extracted in the previous stage into a traceable scalar.

Draw inference result: The output of $\mathcal{F}_{Fc}$, denoted as Z, is the final inference result of the entire coded computation process. It represents a synthesis of the complex interactions and patterns inferred through both the convolutional and fully connected segments of the neural network.

B. Objective

To evaluate the effectiveness of our model, especially in the context of erasure-coded computation, we focus on several key metrics. These include the accuracy of the reconstructed outputs, the efficiency of the encoding and decoding processes, and the overall computational performance in terms of speed and resource utilization. Our experiments, detailed in the subsequent sections, demonstrate the robustness of our approach, particularly in scenarios where data redundancy and recovery are critical.

In this work, we aim to explore the structure of the encoder and decoder, so that the coded computation can be applied to nonlinear $\widehat{\mathcal{F}}_{Conv}(X)$ computation. With this approach, our proposed framework can improve the resilience of distributed inference systems, where device failure or straggler issues can impose significant challenges.

III. Encoder and Decoder

In this section, we aim to design an encoder $\mathcal{E}$ and a decoder $\mathcal{D}$ that can accurately reconstruct the inference output of the conventional segment $\mathcal{F}_{Conv}$ given an arbitrary subset of k computing devices which completed their inference processes. Inspired by [13], we propose to use neural networks to develop the encoders and decoders. Different from [13], we design two pairs of encoder and decoder architectures (with different properties) to better adapt to the distributed inference systems.

In the following, we first propose the encoder and decoded architectures. Then, we present the training process.

TABLE I: Architecture for Encoder

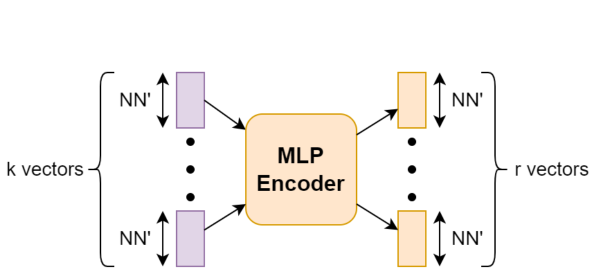
MLPEncoder	ConvEncoder
FC: $kNN' \times kNN'$	Kernel: 3×3, dilation 1
FC: $kNN' \times rNN'$	Kernel: 3×3, dilation 1
	Kernel: 3×3, dilation 2
	Kernel: 3×3, dilation 4
	Kernel: 3×3, dilation 8
	Kernel: 3×3, dilation 1
	Kernel: 1×1, dilation 1

A. Encoder Architecture

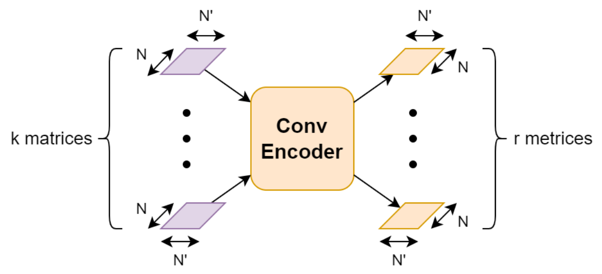
We introduce two encoders based on MLP and CNN, which are called *MLPEncoder* and *ConvEncoder*, respectively. Their corresponding inputs and outputs are presented in Figure 2, and their architectures are given in Table I.

1) MLPEncoder Architecture: We propose a 2-layer MLP encoder architecture, called *MLPEncoder*. Let N and N' denote ??. In this architecture, each $N \times N'$ data input is flattened into a one-dimensional vector, as illustrated in Figure 2a. The k flattened vectors from split input $X_1, X_2, \ldots, X_k$ are concatenated to form a single kNN'-length input vector for *MLPEncoder*. The first fully-connected layer of the MLP produces a kNN'-length hidden vector. The second fully-connected layer produces an rNN'-length output vector, which represents the r redundant inputs. Each layer used in *MLPEncoder* is outlined in Table I. Note that MLPEncoder exploits fully-connected layers, so a small number of layers (e.g., two layers) is sufficient for the computation of arbitrary combinations of the kNN' input features. This makes MLPEncoder easy for implementation and xxx. In Section IV, we show that MLPEncoder is efficient under ?? scenarios.

2) ConvEncoder Architecture: While effective and simple enough for some scenarios, the large number of parameters in *MLPEncoder* can consume a large amount of computational resources and lead to over-fitting. Therefore, *ConvEncoder* is introduced to address this problem. As shown in Figure 2b, the input of *ConvEncoder* contains two-dimensional matrices of size $N \times N'$. Each layer used in ConvEncoder is outlined in Table I. Note that the inputs consist of multiple channels. We design the encoders to process each channel independently. In this process, when an encoder receives k images as input, each having c channels, it operates on each channel of these images in a separate manner. As a result, the encoder outputs r images, and each of these images retains the same c channels as the original inputs. This independent channel-wise processing ensures that the encoder effectively maintains the integrity of the multi-channel structure throughout the encoding process.

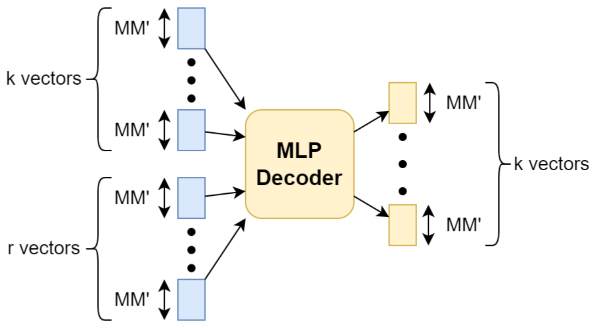


(a) MLPEncoder: Inputs are straighten into a one-dimensional vector with the length of kNN', and the length of output is rNN'.

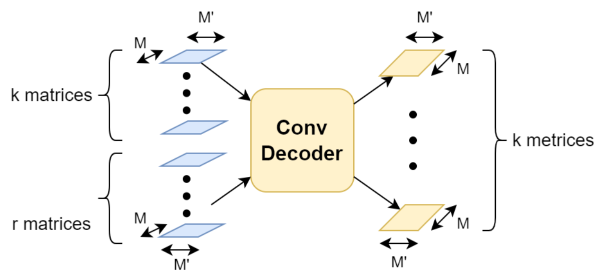


(b) ConvEncoder: Inputs are k matrices of size $N \times N'$, and the outputs are r matrices of size $N \times N'$.

Fig. 2: Inputs and outputs of encoders.



(a) MLPDecoder: Inputs are straighten into a one-dimensional vector with the length of nMM', and the length of output is kMM'.



(b) ConvDecoder: Inputs are n matrices of size $M \times M'$, and outputs are k matrices of size $M \times M'$.

Fig. 3: Inputs and outputs of decoders.

B. Decoder Architecture

In corresponding to MLPEncoder and ConvEncoder, we introduce two decoders *MLPDecoder* and *ConvDecoder*, respectively. Their associated inputs and outputs are presented in Figure 3, and their architectures are given in Table II. The design of these decoder architectures needs to address two questions: 1) How to represent the unavailable outputs of the computing devices (i.e., the outputs of the computing devices that did not complete their inference processes)? 2) What is the specific neural network structure? We address these two questions respectively as follows.

1) Representing Unavailability: The decoder is designed to be inputted $(k + r)$ feature maps, each having a size of $M \times M'$, where M and M' denote ??. These inputs of $(k + r)$ feature maps correspond to the outputs determined by the computing devices using the function $\mathcal{F}_{Conv}$, i.e., $\mathcal{F}_{Conv}(X_1), \dots, \mathcal{F}_{Conv}(X_n)$. In scenarios with device failure and stragglers such that some outputs are unavailable, we propose to let the decoder substitute those unavailable outputs with images filled with zeros. Then, the decoder outputs k vectors $\widehat{\mathcal{F}}_{Conv}(X_1), \dots, \widehat{\mathcal{F}}_{Conv}(X_k)$, each providing an approximate reconstruction of the respective function output that might be unavailable.

2) Decoder Architecture: For the MLPDecoder decoder, we adopt a 3-layer MLP architecture, detailed in Table II. The inputs to the decoder are the flattened outputs of the function $\mathcal{F}_{Conv}$. The decoder comprises three fully-connected layers,

TABLE II: Architecture for Decoder.

MLPDecoder	ConvDecoder
FC: $nMM' \times kMM'$	Kernel: 3×3, dilation 1
FC: $kMM' \times kMM'$	Kernel: 3×3, dilation 1
FC: $kMM' \times kMM'$	Kernel: 3×3, dilation 2
	Kernel: 3×3, dilation 4
	Kernel: 3×3, dilation 8
	Kernel: 3×3, dilation 1
	Kernel: 1×1, dilation 1

with ReLU activations as the final layer. For the ConvDecoder decoder, ...

C. Training Process

For training purposes, the encoder and decoder are trained using the same dataset that originally trained $\mathcal{F}_{Conv}$.

1) Data sample collection: Each training sample for the encoder and decoder is derived from a single input in the training dataset. It can be represented as a pair $(X, \mathcal{F}_{Conv}(X))$, where X corresponds to an original input (before input splitting).

2) Training process: The training cycle contains a forward phase and a backward phase. In the forward phase, as in Figure 1, input X is split into k pieces based on the input splitting idea in Section II-A. The k split inputs $X_1, X_2, \dots, X_k$ are then channeled through the encoder, generating r redundant inputs $X_{k+1}, X_{k+2}, \dots, X_{k+r}$. Subsequently, all n inputs, both split and redundant inputs, are processed through $\mathcal{F}_{Conv}$. The outputs $\mathcal{F}_{Conv}(X_1), \dots, \mathcal{F}_{Conv}(X_n)$ are then input into the decoder $\mathcal{D}$, with a select number of these outputs rendered unavailable, represent as i, $0 \leq i \leq r$. The decoder $\mathcal{D}$ is tasked with reconstructing approximations of the outputs of $\mathcal{F}_{Conv}(X_1), \ldots, \mathcal{F}_{Conv}(X_k)$. Finally, these k inference outputs are merged to form $\widehat{\mathcal{F}}_{Conv}(X)$. The backward pass employs a pre-chosen loss function, back-propagating through $\mathcal{D}$, $\mathcal{F}_{Conv}$, and $\mathcal{E}$ while updating parameters of $\mathcal{D}$ and $\mathcal{E}$ only. This training method is applicable to any function $\mathcal{F}_{Conv}$ that is numerically differentiable.

3) Loss function: We opt for the mean-squared error between the reconstructed output $\widehat{\mathcal{F}}_{Conv}(X)$ and the original output $\mathcal{F}_{Conv}(X)$, which can be regarded as the expected result without device failure and stragglers. This loss function is adequately versatile for numerous functions $\mathcal{F}_{Conv}$.

IV. Experimental Results

We evaluate the accuracy of our proposed encoder and decoders under the scenario in which $\mathcal{F}_{Conv}$ is the convolutional segment of a CNN for the image classification task. We conduct experiments with MNIST dataset.

A. Experimental Settings

Base Models: The base model is the variant of LeNet-5, which consists of two convolutional layers following by ReLU and pooling for each, and three fully-connected layers (dimensions 256×120, 120×84, 84×10) with ReLU after each layer except the last. Table III details the accuracy for base model on given task, as previously discussed in Section III.

TABLE III: Original Accuracy for Base Model.

Base Model	MNIST
LeNet-5	97.93

Hyperparameters: The experiments, encompassing three distinct scenarios outlined in Table IV, were conducted using minibatches, each containing 64 samples. In each scenario, specific values for the number of split data segments k, redundant data r, and the number of unavailable segments l are predetermined and adhered to. For the training process, both the encoder and the decoder are trained simultaneously using the Adam optimization algorithm. This training is characterized by a learning rate set at 0.001 and an L2 regularization factor of 10^{-5}. Such hyperparameter choices are designed to optimize the learning process and ensure the effectiveness and efficiency of the encoder-decoder training under the given experimental conditions.

TABLE IV: Hyperparameter Setting.

Dataset	Basemodel	k	r	l
MNIST	LeNet-5	2	1	1
		4	1	1
		4	2	2

B. Results

The evaluation results for various scenarios, as detailed in Table V, demonstrate the effectiveness of our learning-based erasure-coded computation approach. Notably, these results indicate that the approach is particularly successful in simple tasks when paired with simple base models. Building on this foundation, we plan to extend our investigations to more complex tasks and a broader range of base models. This expansion aims to thoroughly assess the versatility and robustness of our method across a diverse spectrum of challenges and computational models.

TABLE V: Accuracy for Coded Inference

k	r	l	Acc.	degrade
2	1	1	90.56	7.37
4	1	1	96.85	1.08
4	2	2	91.79	6.14

V. Conclusion

In this work, we ..... For future work, ... Future work will focus on exploring the applicability of our approach to more intricate tasks and advanced models, with the goal of establishing a comprehensive understanding of its potential and limitations in varied computational contexts.

References

[1] Neha Sharma, Vibhor Jain, and Anju Mishra. An analysis of convolutional neural networks for image classification. *Procedia Computer Science*, 132:377–384, 2018. International Conference on Computational Intelligence and Data Science.

[2] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[3] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, 2021.

[4] Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. Deepslicing: Collaborative and adaptive cnn inference with low latency. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2175–2187, 2021.

[5] Samson B. Akintoye, Liangxiu Han, Huw Lloyd, Xin Zhang, Darren Dancey, Haoming Chen, and Daoqiang Zhang. Layer-wise partitioning and merging for efficient and scalable deep learning, 2022.

[6] Guiying Li, Chao Qian, Chunhui Jiang, Xiaofen Lu, and Ke Tang. Optimization based layer-wise magnitude-based pruning for dnn compression. In *International Joint Conference on Artificial Intelligence*, 2018.

[7] Feng Xue, Weiwei Fang, Wenyuan Xu, Qi Wang, Xiaodong Ma, and Yi Ding. Edgeld: Locally distributed deep learning inference on edge device clusters. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 613–619, 2020.

[8] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2018.

[9] Qian Yu, Netanel Raviv, Jinhyun So, and Amir Salman Avestimehr. Lagrange coded computing: Optimal design for resiliency, security and privacy. In *International Conference on Artificial Intelligence and Statistics*, 2018.

[10] Mahdi Soleymani, Mohammad Vahid Jamali, and Hessam Mahdavifar. Coded computing via binary linear codes: Designs and performance limits. *IEEE Journal on Selected Areas in Information Theory*, 2:879–892, 2021.

[11] Ankur Mallick and Gauri Joshi. Rateless sum-recovery codes for distributed non-linear computations. In *2022 IEEE Information Theory Workshop (ITW)*, pages 160–165, 2022.

[12] Saurav Prakash, Sagar Dhakal, Mustafa Riza Akdeniz, Yair Yona, Shilpa Talwar, Salman Avestimehr, and Nageen Himayat. Coded computing for low-latency federated learning over wireless edge networks. *IEEE Journal on Selected Areas in Communications*, 39(1):233–250, 2021.

[13] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Learning-based coded computation. *IEEE Journal on Selected Areas in Information Theory*, 1(1):227–236, 2020.