Learning-Based Non-Linear Erasure Code for Distributed Inference

Supervisor: Tang Ming, Graduate Student: Liu Xing,
Team Member: Wang Haoyu, Xu Chunhui, Liu Yiyu

***Abstract*—The objective of this project is to explore the scalability of the learning-based coded computation framework on the basis of replication of the research results of the paper "Learning-Based Coded Computation" [2], which integrates machine learning with coded computation to investigate the resilience of learning-based coded computation in non-linear computations. We will imitate this framework to build our own learning based coded computation framework in distributed scenario that is applicable to our tasks, and to further explore the possibility of enhancing the performance of the framework. Our primary focus will be on the image classification problem, attempting to transform the data flow and network structure of the framework using distributed strategies, thus to reduce hardware dependencies and cost of time, and to improve the accuracy of recoveries under unavailability. Finally, We aim to investigate the potential of learning-based non-linear coded computation through above process.**

***Index Terms*—Coded computation, Distributed Learning, Machine Learning, DNN.**

I. Introduction

DISTRIBUTED learning uses distributed technology in machine learning, enabling multiple computing nodes to collaborate to complete a learning or inference task. In distributed learning, data and computational tasks are distributed across multiple nodes. Each node independently processes a part of the data and collaborates through communication and coordination to learn model parameters or complete inference. This approach is commonly used in large-scale datasets or machine learning tasks requiring substantial computational resources (e.g., deep learning). Distributed learning can accelerate model training, improve performance, and allow handling of large datasets while reducing the burden on individual computing nodes.

Coded computation is a fault-tolerant computational method often used in distributed systems. In coded computation, data is split into multiple chunks and encoded to create redundant data blocks, enabling the restoration of original data when it becomes unavailable. This coding technique is commonly known as Erasure Coding or Error-Correcting Code. Coded computation improves data security and reliability and can reduce the cost of data storage and transmission.

The recent development of artificial intelligence has driven intelligent applications at the network edge. In this scenario, distributed technology becomes a good means to enhance computational security and efficiency. Due to heterogeneity in computing capabilities, hardware resources, network conditions, and device architecture in typical distributed computing environments, distributed inference often faces unavailability, leading to delays. Coded computation has recently been revitalized as a potential approach to alleviate the effects of slowdowns and failures in such systems while using fewer resources than replication-based approaches [3]. However, traditional coded computation has limited scalability, raising the question: Can distributed learning be integrated into coded computation to provide broader scalability?

II. Problem Review for Last Report

A. Related Works

Since the concept of coded computation was proposed, the coded calculation of nonlinear functions has always been a difficulty in the field of coded computation research. Most current research has refined its goals and focused on specific areas' coded computation problems. There are different ideas for nonlinear coded computation. For example, decomposing complex nonlinear computation into multiple simple nonlinear computation[4], using federated learning to share data between distributed devices to achieve redundant computing[7], etc.

B. Expression to Evaluate

In last presentation, we have the expression to evaluate accuracy:

$$A_O = (1 - f)A_a + fA_d$$

Where A_a is accuracy of base model outputs is available and A_d when these outputs are unavailable. f is the fraction of base model outputs are unavailable.

But in our practice, there are some differences: We can't define the f, in coded computation model, there should be as few devices as possible and in the shortest time possible to reconstruct the results, so f usually is fixed by k and r. So this expression is not very applicable.

III. Related Work

A. Learning-Based Coded Computation

1) Introduction: This paper, published in 2020, is the first work to leverage learning for coded computation.

The paper points out that coded computation has the potential to enhance resilience against slowdowns and failures in distributed computing systems. However, existing coded computation methods have low support for nonlinear computation. Against this backdrop, they propose a learning-based coded computation framework to overcome challenges in performing coded computation on general nonlinear functions.

The results show that: a) Using machine learning within the coded computation framework can extend the range of coded computation, providing resilience for more general nonlinear computations. b) Learning-based coded computation can accurately reconstruct outcomes unavailable from widely deployed neural networks used in various inference tasks like image classification, speech recognition, and object localization.

2) Traditional Coded Computation vs Learning-Based Coded Computation: Common distributed computing environments are prone to unavailability (e.g., slowdowns and failures), leading to delays. For such unavailability, coded computation provides new resilience: In traditional coded computation, inputs for computation are first encoded, and after computation, if some outputs are partially unavailable, coded computation can recover these unavailable parts using the encoded inputs, thus improving the performance of distributed computation.

However, traditional coded computation research has focused on linear computations. They either cannot support nonlinear computations or only support a limited subset of nonlinear computations, requiring high resource overhead. This severely limits the application scenarios for coded computation. Therefore, although coded computation is a promising technique to mitigate slowdowns and failures, traditional coded computation is not sufficient to provide resilience for a broader range of nonlinear computations.

Many machine learning models are complex nonlinear functions. The recent advancements in machine learning in complex tasks (such as image classification and natural language processing) suggest its potential to help overcome challenges in encoded computation of nonlinear functions. In this context, this paper makes the following contribution: It proposes a learning-based coded computation framework that uses neural networks as encoders and decoders to learn to perform coded computation on nonlinear functions, effectively reconstructing the unavailable outputs.

3) Architecture: The framework of their work is as follows. Consider the following inference task: There is an inference function $\mathcal{F}$, referred to as the base model, with copies on several devices. Given k inputs $X_1, X_2, \ldots, X_k$, the goal is to compute $\mathcal{F}(X_1), \mathcal{F}(X_2), \ldots, \mathcal{F}(X_k)$. The encoder $\mathcal{E}$ receives k original inputs $X_1, X_2, \ldots, X_k$ and produces r parity inputs $P_1, P_2, \ldots, P_r$. Subsequently, all these $k+r$ original and parity inputs are sent to the copies of $\mathcal{F}$, each computing the respective results. Finally, in the case of up to r unavailable outputs (slow or failed), all k outputs $\mathcal{F}(X_1), \mathcal{F}(X_2), \ldots, \mathcal{F}(X_k)$ are recovered.

In practice, the base model $\mathcal{F}$ is a neural network used for image classification, and this paper's work simulated a situation with $k = 2$ and $r = 1$.

B. EdgeLD: Locally Distributed Deep Learning Inference on Edge Device Clusters

1) Introduction: In edge intelligence application scenarios, due to the difficulty for edge devices to handle the computational and memory burden of traditional DNNs, implementing distributed DNNs becomes reasonable. This paper presents EdgeLD, a distributed inference framework based on DNNs. It achieves distributed inference by distributing the computational load of convolutional networks across multiple devices. Beyond the standard layer-wise mode, it adopts layer fusion optimization to reduce the communication overhead of exchanging intermediate data between devices. Additionally, the paper proposes a model partition scheme to balance the workload and runtime of the model.

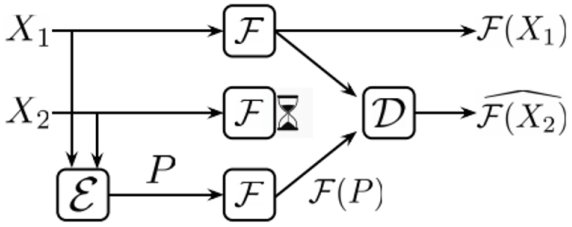

Fig. 1. Example of coded computation with k = 2 original units and r = 1 parity units.

2) Methods:

- Common Model Partitioning Shortcomings
  The heavy resource consumption of DNNs leads to resource strain. Common solutions include Model Compression and Cloud Offloading. Model compression techniques, such as weight pruning and knowledge distillation, improve processing efficiency but result in the loss of the original model's complexity and precision [5]. Cloud Offloading, which decouples DNN computation workload according to execution order and offloads compute-intensive tasks to powerful cloud servers, may suffer from excessive transmission delays due to unstable network connections [6]. Considering these limitations, this paper proposes a new model partitioning method.
- Task Distribution for Devices
  In a cluster of devices, nodes are categorized into two types: Group Leaders (GL) and Follower Nodes (FNs). GL nodes are responsible for distributing the computational workload and delegating tasks to several FNs. Each node independently computes and then aggregates the results back to the GL node.
- Workload Partitioning for Convolutional Layers
  Convolutional layers consume a significant portion of computational resources and time. Therefore, partitioning the computational tasks of convolutional layers into distributed system nodes is considered. The paper introduces One-dimensional Computation&Bandwidth-based Partition (OCBP). Compared to the previous Biased One-Dimensional Partition (BODP) [5], OCBP considers the computational resources and network conditions of heterogeneous nodes.
- Inter-Device Data Interaction and Fusion Layer
  The paper employs a layer-wise parallel computation mode, where each FN node computes only its relevant part of the task, then aggregates it to the GL. However, this inter-device communication can reduce performance. Therefore, the paper also adopts a fused-layer approach to

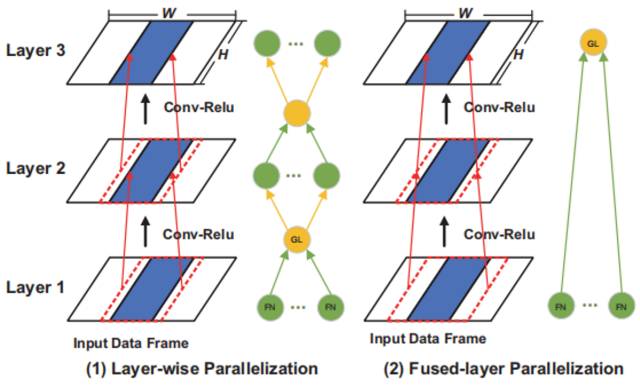

Fig. 2. Layer-wise and fused-layer partitioning strategies.

mitigate this issue. In this approach, consecutive convolutional layers are packaged into fused blocks, calculated as a whole, eliminating the need for inter-device data interaction between adjacent layers, with data transfer occurring only at the input and output of the fused block. This is feasible because of the inherent property of convolutional layers: each layer's results are derived from corresponding points and surrounding points in the previous layer, allowing calculations related to a position in a layer to be mapped upwards.

C. Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices

1) Introduction: In distributed inference, the partitioning of network architecture and workload significantly impacts the effectiveness of distributed inference. In this context, this paper analyzes the trade-offs of splitting inference tasks on lightweight edge devices which includes partitioning workloads of convolutional layers and the way of parallel computing. This paper further proposes a runtime adaptive CNN acceleration framework, which implements a dynamic algorithm for workload distribution in consideration of device capabilities, network conditions, and communication overhead. By fusing convolutional layers and applying this adaptive dynamic planning algorithm to arrange workloads, it provides an effective way to balance the workload of inference in the distributed scenario.

2) Methods:

- Partitioning Methods for Convolutional Layer Computational Load
  Partitioning the computational load of convolutional layers can be done by channel partitioning or spatial partitioning.
  In channel partitioning, each filter in the convolutional layer generates a feature map. These feature maps can be partitioned along the channel dimension and mapped to corresponding devices, with each device computing only a subset of the feature maps. In this mode, the communication cost associated with synchronizing network parameters is reduced (effective during model training),

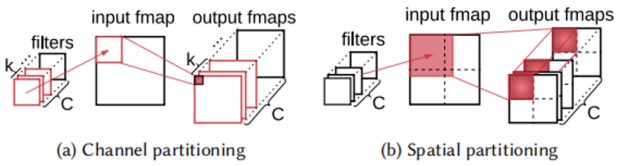

Fig. 3. Examples of parallelizing a 2D convolution layer note that, each filter and its corresponding input/output feature map have the same channel size and are not shown in the plots).

but the need to replicate the entire input image across all devices leads to additional communication costs, often making it less practical for inference tasks compared to spatial partitioning.
In spatial partitioning, feature maps are derived by spatially splitting, and are assigned to multiple nodes. In this mode, each node only needs to receive a part of the input image, with each device having a complete neural network. Since computations related to a specific position in a layer can be mapped upwards in convolutional layers, the input partitions are not entirely disjoint, necessitating consideration of redundant computation and inter-device data interaction issues.

- Trade-offs in Parallel Computation
  There are two strategies for implementing parallel computation: Layer-wise or Fused-layer.
  In the Layer-wise strategy, the computation of each layer is determined based on its type, feature map shape, filter size, etc., and each layer is parallelized individually. After computing each layer, the results from all nodes are merged to form the output for that layer, and then the next layer is partitioned and computed in parallel. This strategy requires a host node to continuously distribute and collect parallel computing tasks.
  In the Fused-layer strategy, several consecutive convolutional layers are packaged into a fused block, participating in parallel computation as a whole, similar to the scenario in EdgeLD [8]. In a fused block, the output of one layer is directly sent to the corresponding input of the next layer, avoiding frequent task distribution and collection, thus reducing device communication costs. However, due to the upward mapping nature of convolutional layer computations, there is some overlap in input partitions, leading to additional redundant computation. The more layers in a fused block, the more communication cost it can reduce, but the cost of redundant computation due to overlapping input partitions also increases. Overall, the Fused-layer method has positive optimization potential. Careful use of fused-layer parallelism on top of layer-wise parallelism has shown practical value in implementations, providing insights into our project structure [8][10][1].

D. Some Other Work

In addition to the papers mentioned above, several other methods related to distributed strategies have been proposed.

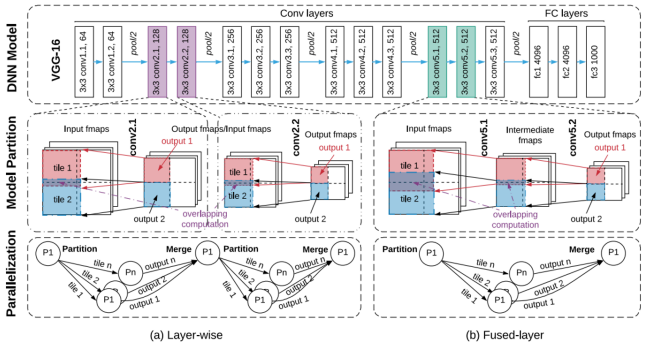

Fig. 4. Illustration of layer-wise parallelization (a) vs layer fusion (b)

A significant commonality among them is the emphasis on partitioning the workload in distributed computing, especially the computational load of convolutional layers. These methods can generally be categorized into two aspects: a. Partitioning methods from the perspective of neural network structure and data flow, and b. Partitioning algorithms considering device heterogeneity and node operational conditions.

In terms of partitioning methods from the perspective of neural network structure and data flow, considering the substantial computational load of convolutional layers, many methods start by optimizing these layers. Based on layer-wise parallel computation, they implement a certain degree of fused-layer parallel computation and carefully explore the trade-offs and compromises this method offers in enhancing computational performance [8][10].

Regarding partitioning algorithms that consider device heterogeneity and node operational conditions, most methods divide nodes into master and slave categories. Here, the master node is responsible for the overall control of the workload, implementing various algorithms for rational workload distribution, and then completes distributed computing tasks in conjunction with slave nodes. In addition to dynamic workload distribution algorithms that consider device computational resources and network conditions, other algorithms have been proposed. For example, running a device profiling algorithm to assess device computational resources and then aggregating the results at an analysis engine to generate a partitioning plan [9]; or using a work-stealing algorithm to allow nodes with spare computational resources to take over tasks from nodes under computational stress[1], among others.

IV. Preliminary

Building upon the coded computation model presented in [2], our study introduces a novel variation in the neural network context. As depicted in Figure 5, erasure-coded computation model introduces an encoder $\mathcal{E}$, a decoder $\mathcal{D}$. A function $\mathcal{F}_{Conv}$ denotes the convolutional segment of CNN and a function $\mathcal{F}_{Fc}$. In our scenario, the input X is initially split into k distinct parts $X_1, X_2, ..., X_k$. This division is crucial for our proposed method as it allows for a more granular approach to encoding and redundancy creation.

Each part of the divided input undergoes the encoding process, resulting in r additional redundant segments,

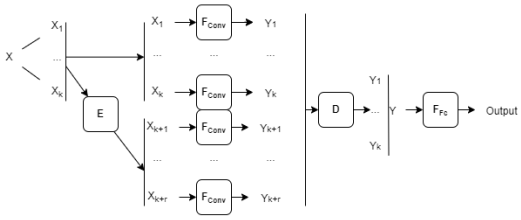

Fig. 5. Overall framework for non-linear erasure-coded computation.

$X_{k+1}, ..., X_{k+r}$. These segments, alongside the original k parts, form the $k + r$ inputs for next computation. This results in a total of $n = k + r$ encoded segments, denoted as $X_1, X_2, ..., X_k, ..., X_n$, where n represents the number of distributed devices involved in the computation. These n segments are then prepared as inputs for the subsequent computational stages.

The n inputs, $X_1, X_2, ..., X_n$, are individually processed by copies of a non-linear function $\mathcal{F}_{Conv}$, emblematic of convolutional segments found in convolutional neural networks (CNNs). The function $\mathcal{F}_{Conv}$ applies complex transformations to each input, thereby introducing the requisite depth and intricacy into the computation. These transformations are essential for capturing and encoding the nuanced patterns and relationships present in the data.

The output from each instance of $\mathcal{F}_{\text{Conv}}$ constitutes a transformed data point, encapsulating the features and characteristics extracted from its corresponding input following non-linear transformation. These outputs represent the features that have been both enriched and encoded, thus preparing them for the subsequent phase in the computational process. Subsequently, these outputs are inputted into the decoder $\mathcal{D}$, resulting in k predicted convolutional outputs $\hat{Y}_1, \hat{Y}_2, \ldots, \hat{Y}_k$. Then, these outputs are amalgamated into a single output $\hat{Y}$, represented as $\widehat{\mathcal{F}_{\text{Conv}}(X)}$, symbolizing the comprehensive output of the convolutional process.

Following the convolutional processing, output $\hat{Y}$ is then subjected to the second function, $\mathcal{F}_{\text{Fc}}$, which denotes the fully connected segment of the CNN. This phase of computation is crucial for integrating the high-level features extracted in the previous stage. $\mathcal{F}_{\text{Fc}}$ performs a series of linear transformations, mapping the enriched feature set into a final output space that is more suitable for downstream tasks such as classification or regression.

The output of $\mathcal{F}_{\text{Fc}}$, denoted as Z, is the culmination of the entire coded computation process within our neural network model. It represents not just the transformed data, but a synthesis of the complex interactions and patterns learned through both the convolutional and fully connected layers of the network.

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.

The overall loss $\mathbf{Loss}_{overall}$ between $\widehat{\mathcal{F}_{Conv}(X)}$ and $\mathcal{F}_{Conv}(X)$ can be written in the form of the weighted average of the loss when i device's outputs are unavailable, where $0 \leq i \leq r$.

$$\mathbf{Loss}_{overall} = \frac{1}{r}\sum_{i=0}^{r} \alpha_i \mathbf{Loss}_i$$

α_i is the coefficient of $\mathbf{Loss}_i$, which represents some function related to k and r but we do not confirm now.

$$\alpha_i = f(k, r)$$

In summary, our contribution extends the coded computation model by incorporating a dual-stage processing paradigm within CNNs, leveraging both $\mathcal{F}_{\text{Conv}}$ and $\mathcal{F}_{\text{Fc}}$. This approach not only enhances the model's ability to handle complex datasets but also improves its resilience in distributed computing environments, where data loss or corruption can be significant challenges.

V. Non-linear Erasure-Coded Computation

In this section, we introduce our approach to non-linear erasure-coded computation. Recall from Section III, and as illustrated in Figure 5, our coded computation framework includes 4 core components: the convolutional segment function $\mathcal{F}_{Conv}$, the fully connected segment function $\mathcal{F}_{Fc}$, the encoder $\mathcal{E}$, and the decoder $\mathcal{D}$. The aim here is to design an encoder $\mathcal{E}$ and a decoder $\mathcal{D}$ that can accurately reconstruct the original output for the function $\mathcal{F}_{Conv}$. Leveraging the recent successes of neural networks in various tasks, we utilize these networks to develop our encoders and decoders.

We next describe detail the training methodologies for the encoders and decoders, along with the specific neural network architectures utilized in learning encoder and decoder.

A. Training Process

The primary goal of our training protocol is to develop neural network encoders and decoders capable of effectively reconstructing unavailable outputs from the function $\mathcal{F}_{\text{Conv}}$. This function, which acts as the convolutional segment of a CNN, remains unchanged during the training phase. For training purposes, when $\mathcal{F}_{\text{Conv}}$ is utilized, the encoder and decoder are trained using the same dataset that originally trained $\mathcal{F}_{\text{Conv}}$.

Each training sample for the encoder and decoder is derived from a single input in the training dataset, which is split into k smaller segments. The size of each segment is calculated to be slightly larger than the original input size divided by k, a determination made in accordance with the structural requirements of $\mathcal{F}_{\text{Conv}}$. Consequently, the training data can be represented as pairs $(X, \mathcal{F}_{\text{Conv}}(X))$, where X embodies the original inputs.

A typical training cycle, inclusive of a forward and backward pass, is illustrated in Figure 5. In the forward phase, k split inputs $X_1, X_2, \ldots, X_k$ are channeled through the encoder, generating r redundant inputs $X_{k+1}, X_{k+2}, \ldots, X_{k+r}$. Recall that $n = k + r$ means the number of devices. Subsequently, all n inputs, both split data and redundancy, are processed through $\mathcal{F}_{\text{Conv}}$. The outputs $\mathcal{F}_{\text{Conv}}(X_1), \ldots, \mathcal{F}_{\text{Conv}}(X_n)$ are then input into the decoder $\mathcal{D}$, with a select number of these outputs rendered unavailable (as detailed in following section). The decoder $\mathcal{D}$ is tasked with reconstructing approximations of the outputs of $\mathcal{F}_{\text{Conv}}(X_1), \ldots, \mathcal{F}_{\text{Conv}}(X_k)$. Finally, these k inference outputs are merged to form $\widehat{\mathcal{F}_{\text{Conv}}(X)}$. The backward pass employs a selected loss function, back-propagating through $\mathcal{D}$, $\mathcal{F}_{\text{Conv}}$, and $\mathcal{E}$ with updating parameters of $\mathcal{D}$ and $\mathcal{E}$ only. This training method, which includes back-propagation through $\mathcal{F}_{\text{Conv}}$, is adaptable to any function that is numerically differentiable.

Various loss functions are viable for training encoders and decoders. We opt for the mean-squared error between the reconstructed output $\widehat{\mathcal{F}_{\text{Conv}}(X)}$ and the original output $\mathcal{F}_{\text{Conv}}(X)$, which would be the expected result in the absence of slowdowns or failures in the base model. This loss function is adequately versatile for numerous functions $\mathcal{F}_{\text{Conv}}$. Function-specific loss functions, like cross-entropy in image classification tasks, can also be applied.

The subsequent section will detail the specific encoder and decoder architectures adopted in this methodology. To illustrate, we will describe these architectures considering $\mathcal{F}_{\text{Conv}}$ as the convolutional segment of a neural network image classifier. Here, original data input X is an $a \times a$ pixel image, each split data input X_i is an $a \times a'$, and the corresponding output $\mathcal{F}_{\text{Conv}}(X)$ is an $b \times b'$ feature map emanating from the final layer of $\mathcal{F}_{\text{Conv}}$.

B. Encoder Architecture

We only consider one neural network architecture for learning the encoder at the first stage of out work. Table I outlines each layer of the neural network encoders we propose that make use of Multi-layer Perceptron (MLP).

TABLE I
Architecture for Encoder.

Layer	MLPEncoder
1	FC: $kaa' \times kaa'$
2	FC: $kaa' \times raa'$

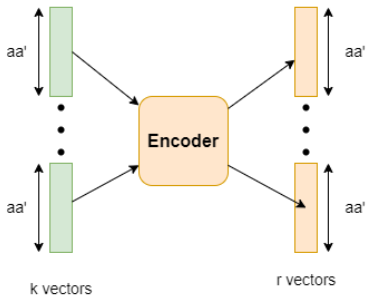

Fig. 6. Inputs and outputs of MLPEncoder

The simple 2-layer MLP encoder architecture is proposed, which is called *MLPEncoder*. Under this architecture, each $a \times a'$ data input is flattened into a one-dimensional vector, as illustrated in Figure 6. The k flattened vectors from split input $X_1, X_2, \ldots, X_k$ are concatenated to form a single kaa'-length input vector to *MLPEncoder*. The first fully-connected layer of the MLP produces a kaa'-length hidden vector. The second fully-connected layer produces an raa'-length output vector, which represents the r redundant inputs. Each layer used in *MLPEncoder* is outlined in Table I.

The fully-connected layers of MLP allows for computation of arbitrary combinations of the kaa' input features using few layers. While effective and simple enough for some scenarios, the high parameter count of MLP can assume a lot of calculating resource and lead to over-fitting. We will try more architectures in our future work.

For inputs consisting of multiple channels, the encoders we have described are designed to process each channel independently. In this setup, 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 crucially, each of these images retains the same c channels as the original input. This independent channel-wise processing ensures that the encoder effectively maintains the integrity of the multi-channel structure throughout the encoding process.

C. Decoder Architecture

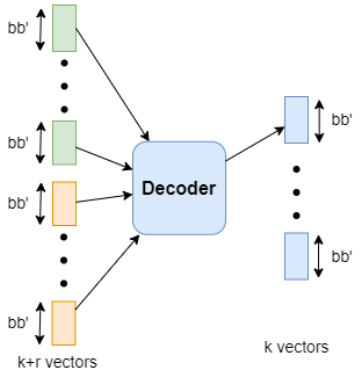

Fig. 7. Inputs and outputs of MLPDecoder. No unavailable input is shown in this figure

Figure7 provides a comprehensive high-level overview of the decoder's architecture. Within this architecture, there are two critical design elements that are pivotal: firstly, how the unavailable outputs from the base model are represented at the input layer of the decoder; and secondly, the specific neural network architecture that is employed for the decoder. These key design choices are instrumental in determining the efficacy and functionality of the decoder in our computational framework.

1) Representing Unavailability: The decoder is designed to accept $(k+r)$ feature maps, each of size $b \times b'$, as input. These inputs include the outputs from the function $\mathcal{F}_{\text{Conv}}$, denoted as $\mathcal{F}_{\text{Conv}}(X_1), \ldots, \mathcal{F}_{\text{Conv}}(X_n)$. In scenarios where some of these inputs are unavailable, the decoder substitutes them with images filled entirely with zeros. However, if a zero image is a plausible output for the given function $\mathcal{F}_{\text{Conv}}$, an alternate representation is utilized to indicate unavailability. As a result, the decoder outputs k vectors $\widehat{\mathcal{F}_{\text{Conv}}(X_1)}, \ldots, \widehat{\mathcal{F}_{\text{Conv}}(X_k)}$, each providing an approximate reconstruction of the respective function output that might be unavailable.

2) Decoder Architecture: For the decoder, we have adopted a 3-layer Multi-Layer Perceptron (MLP) architecture, detailed in Table II. The decoder comprises three fully-connected layers, with ReLU activations following all but the final layer. The inputs to the decoder are the flatten outputs of the base model $\mathcal{F}_{\text{Conv}}$.

TABLE II
ARCHITECTURE FOR DECODER.

Layer	MLPDncoder
1	FC: $nbb' \times kbb'$
2	FC: $kbb' \times kbb'$
3	FC: $kbb' \times kbb'$

VI. EXPERIMENTAL RESULTS

We evaluate the accuracy of learned encoder and decoders by focusing on the scenario in which $\mathcal{F}_{\text{Conv}}$ is the convolutional segment of a CNN for image classification task.

A. Setup

a) Implementation: All encoder and decoder architectures were implemented using PyTorch. We assessed the effectiveness of our approach using a simplest dataset: MNIST.

b) 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

c) 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. 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.

TABLE V
ACCURACY FOR ERASURE-CODED COMPUTATION

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

VII. FUTURE WORKS

In our forthcoming research endeavors, we aim to explore a wider array of encoder and decoder architectures. This exploration will not only involve implementing these architectures but also rigorously testing their performance across a diverse range of tasks and computational models. By expanding the scope of our investigations, we intend to assess the adaptability and efficiency of various encoder-decoder configurations in handling different types of tasks, each with its unique set of challenges and requirements. This comprehensive analysis will contribute to a deeper understanding of the potential and limitations of these architectures in varied application scenarios, ultimately guiding the development of more robust and versatile coded computation methodologies.

REFERENCES

[1] A. Gerstlauer, Z. Zhao, and K. M. Barijough. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. November 2018.

[2] J. Kosaian, K. V. Rashmi, and S. Venkataraman. Learning-based coded computation. May 2020.

[3] 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.

[4] 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.

[5] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1396–1401, 2017.

[6] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 751–756, 2017.

[7] 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.

[8] F. Xue, W. Fang, W. Xu, et al. Edgeld: Locally distributed deep learning inference on edge device clusters. 2020.

[9] J. Zhang, L. Yang, Z. Zhou, et al. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. April 2021.

[10] L. Zhou, M. H. Samavatian, A. Bacha, et al. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. Sept 2019.