Coded Edge Computing

Kwang Taik Kim*, Carlee Joe-Wong†, and Mung Chiang*
*Purdue University, †Carnegie Mellon University
Email: *kimkt@purdue.edu, †cjoewong@andrew.cmu.edu, *chiang@purdue.edu

***Abstract*—Running intensive compute tasks across the fifth generation mobile network of edge devices introduces distributed computing challenges: edge devices are heterogeneous in the compute, storage, and communication capabilities; and can exhibit unpredictable straggler effects and failures. In this work, we propose an error-correcting-code inspired strategy to execute computing tasks in edge computing environments, which is designed to mitigate variability in response times and errors caused by edge devices' heterogeneity and lack of reliability. Unlike prior coding approaches, we incorporate partially unfinished coded tasks into our computation recovery, which allows us to achieve smooth performance degradation with low-complexity decoding when the coded tasks are run on edge devices with a fixed deadline. By further carrying out coding on edge devices as well as a master node, the proposed computing scheme also alleviates communication bottlenecks during data shuffling and is amenable to distributed implementation in a highly variable and limited network. Such distributed encoding forces us to solve new decoding challenges. Using a representative implementation based on federated multi-task learning frameworks, extensive performance simulations are carried out, which demonstrate that the proposed strategy offers significant gains in latency and accuracy over conventional coded computing schemes.**

***Index Terms*—edge computing, coding theory, distributed learning**

I. Introduction

As the Internet-of-Things (IoT) begins to be deployed in the fifth generation (5G) mobile network, devices such as smartphones, smart speakers, and wearable hardware will increasingly pervade our lives, appearing in environments ranging from the industrial to residential. These devices will, and indeed many already do, generate increasing amounts of data, e.g., by sensing users' inputs and surrounding information about their environments. This data can be used for applications such as scene analysis, speech recognition, and task-specific assistance by analyzing it at real-time cognitive engines located at both mobile devices (where the data is collected) and cloud servers. However, such an edge computing setting introduces new challenges compared to existing distributed environments such as servers in data centers. Unlike homogeneous datacenter servers, mobile devices located at the network edge or mobile edge clouds have heterogeneous communication, computation, energy, and storage capabilities. Their connectivity topology may also be more complex than that of worker nodes in data centers. Such distributed heterogeneous environments require parallel processing due to highly latency-sensitive resource-intensive (human-in-loop) systems to achieve device learning objectives. Moreover, edge devices likely exhibit unpredictable latency due to various factors such as hardware reliability (e.g., disk failures, battery limits), varying network conditions, shared resource contention, and unbalanced workloads.

To design scalable and robust systems in these edge computing environments, we therefore need to address their unreliability and variable response times. MDS (maximum distance separable)-based coded computing has received considerable attention from the research community to speed up dispersed computing systems by injecting redundancy computations [1]–[8]. It has been shown that the coding approach can speed up machine learning in distributed settings by a multiplicative factor that is proportional to the amount of injected redundancy. For example, several coding techniques have been proposed that compute large linear transforms in a distributed way [9], convolve two long vectors using parallel processors [10], evaluate nonlinear functions by exploiting the modern multicore setups [11], multiply large matrices [12], and enable a hierarchical computing system [13]. However, most of these works assume balanced and i.i.d. data processed at homogeneous worker nodes since they were designed to distribute datasets among many worker nodes in a single data center. Edge computing, however, should deal with system challenges – edge devices are heterogeneous in terms of computing capability, communication resource, battery, etc.

In the light of these system challenges, previous works suffer from a number of performance and scalability problems. The primary limitation is that (n, k)-coded computation schemes based on MDS coding can not recover the desired computation if k coded computations are not collected within the deadline. Since many edge devices may not be able to fully complete or return their computations due to limited battery, limited communication capability, and/or mobility, MDS coding can often fail, forcing the master node to wait longer for additional complete computation results from its workers. In addition, unlike in data center environments with well-understood failure statistics, it is difficult to know what value should be set for k, the number of data partitions, to achieve the best coding performance when the coded computation is distributed among n worker edge devices.

The second limitation is that the previous works have not considered dynamically changing network topologies beyond carrying out coding in the master node. Due to the mobility of wireless edge nodes, their network connectivity could degrade quickly, and some edge nodes could disappear during learning. Thus, the master node does not have control on the structure of the generator matrix used by the worker nodes, as this generator matrix will change if worker nodes disappear.

In this work, we propose to address these two fundamen-

tal limitations by developing error-correcting-code inspired techniques that outperform MDS-based coding techniques for matrix-vector multiplication ($\mathbf{y} = \mathbf{X}\mathbf{w}$), which is a fundamental step for many data analytics algorithms, such as principal component analysis, logistic regression, collaborative filtering, etc. However, designing such techniques poses several research challenges. First, in contrast to common coding theory [14] or network coding settings (see [15], [16], and references therein), the contents of the redundancy data depend on the actual data matrix, but should not depend on the task weight vector, even though the decoding procedure should be performed on the coded computations (i.e., the data multiplied by the weight vector), not the coded data. We thus face decoding challenges. The actual computations can be associated with different coded computations, depending on the task weight vector. This is different from common coding theory or network coding settings where redundancy symbols are uniquely determined by information symbols, which do not allow us to recover unique target computations in our setting.

Second, unlike coded shuffling in cloud computing, it is difficult to track what cached data resides at which edge worker devices. Since the overhead could be high if the master node has to track which previously joined edge devices have what coded data over iterations, it is reasonable for each edge device to make its own coding decision to improve the statistical efficiency of the learning procedure. Even if the master node chooses the coding coefficients for all edge devices, it still has no control over the structure of the generator matrix G, because some coded computations might not be returned due to mobility, battery, or network congestion, which cannot be assumed to be known a priori. The master node then cannot always control how edge devices re-encode the coded data as needed for matrix-vector multiplications, and the reconstructed generator matrix G is arbitrarily determined rather than pre-designed. Thus, a master node needs to decode with an arbitrary generator matrix. However, it is known in coding theory that it is NP-hard to decode a linear code with an arbitrary generator matrix [17].

We provide four key insights into this problem, which serve as building blocks that enable us to design a coded computing strategy that overcomes the uncontrollable edge environment and limits the error of the final machine learning models encountered when stragglers occur. Our first insight is to *treat the difference from the complete computation as additive noise*, instead of flagging unfinished task as erasures. This error model allows us to exploit intermediate results to evaluate the reliability of each computation. Our second insight is to carefully *combine the ideas of quantization, quantized data encoding, and multi-stage computation decoding via modulo operations* to resolve the aforementioned nonunique redundancy symbols. This key methodology is highlighted in Theorem 1 in Section III. Our third insight is to *jointly carry out coding on edge nodes as well as the master node*, instead of carrying out coding on the master side only and then injecting task redundancy to edge nodes, during data shuffling between distributed nodes. In this way, we ensure that only a few coded multicast generates sufficient randomness and alleviate communication bottlenecks. Finally, we *combine a reconstructed generator matrix from control information (consequently the corresponding parity check matrix) with the idea of sphere decoding (allowing low complexity decoding)*. This component of decoding enables to identify candidates for coded computations and recover the actual computations by harnessing reliability.

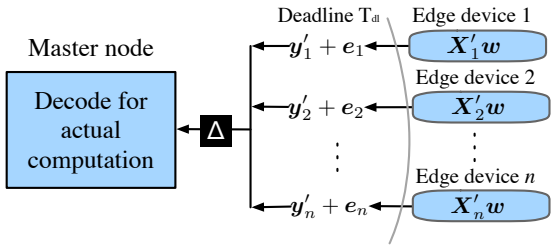


Fig. 1. The system model of coded edge computing.

The rest of the paper is organized as follows. The next section describes the system model that we assume in designing our proposed coded computing strategy. The main contributions of the paper (our proposed coded edge computing strategy) appear in Section III. In Section IV, we evaluate the overall performance of the proposed strategy for matrix-vector multiplications and for federated learning, showing that it consistently outperforms MDS-based coded computing and uncoded computing over a range of straggler settings.

II. System Model

We consider a system model that performs distributed edge computing where (i) the master node in the edge environment executes a data analytics algorithm and (ii) the master node assigns partial replications of the training data $\mathbf{X}$ via coding techniques to many nearby edge devices that may be one or multiple hops away. The edge devices are used to accelerate the multiplication of $\mathbf{X}$ with the task weight vector $\mathbf{w}$ and mitigate the effect of straggling edge devices; with our proposed coding strategy, they can also significantly reduce the errors caused by stragglers. Each device j multiplies a coded subset of the training data $\mathbf{X}'_j$ by $\mathbf{w}$ and returns the (possibly unfinished) coded computations at the deadline T_{dl}. Instead of ignoring the coded computation results of slow edge devices as erasures, while waiting for k fast edge devices like the MDS-coded approach, our proposed strategy *fully utilizes the intermediate computation results* returned within the deadline T_{dl} from all edge devices. We recover the desired output by treating the difference from the optimal solution as additive noise: $\mathbf{e}_j = \mathbf{y}_j - \mathbf{y}^*_j$ for $j = [1 : n]$, where n is the number of edge devices serving the master node. Our system model of coded edge computing is illustrated in Figure 1.

Once the results from the edge users are received, the master node must decode them by reconstructing the associated generator matrix G, which depends on which coded computations are received. Some coded computations might not be returned

due to mobility of edge devices, battery outage, or network congestion. The master node then decodes the erroneous received computations to recover the actual computations.

III. Coded Strategy for Edge Computing

We first provide an overview of our proposed coded strategy for edge computing, as shown in Algorithm 1. We focus on computing a basic matrix-vector multiplication $\mathbf{y} = \mathbf{X}\mathbf{w}$, which is a fundamental step for many data analytics algorithms [1].

The first steps in our strategy are to quantize the available training data $\mathbf{X}$ and weight vector $\mathbf{w}$ at the master node, and then encode the quantized data (lines 1 and 2). The master node then distributes the encoded data and quantized task weight vector to its nearby edge devices (line 3) so that each device can compute its own matrix-vector multiplication (lines 4 and 5).

At deadline T_{dl}, all edge devices return their partially or completely finished computations back to the master node (line 6). We denote these results as $\mathbf{y}'_j + \mathbf{e}_j$ for each edge device j, where $\mathbf{e}_j$ models the error in the device's computation due to not finishing the computations or other error factors. This error can be large; devices that have failed, e.g., lost network connectivity, would not return any computations and we can assume that their computations are entirely unfinished (i.e., $\mathbf{e}_j = -\mathbf{y}'_j$). In practice, we would not know $\mathbf{e}_j$ as we only observe the computed result $\mathbf{y}'_j + \mathbf{e}_j$, not $\mathbf{y}'_j$; however, modeling incompleteness as a computation error allows us to recover a more accurate estimate of $\mathbf{y}$ than standard coding techniques, by leveraging partially finished results.

The master node finally decodes the received coded computations by remapping them into binary forms via modulo 2 operations and carefully combining multi-level stage decoding and sphere decoding (lines 9 to 12), finally returning an estimate $\hat{\mathbf{y}}$ of $\mathbf{X}\mathbf{w}$.

We describe each step of our strategy in detail below.

Algorithm 1 Error-Correcting Data Encoding and Computation Decoding for Edge Computing

Input: Training data $\mathbf{X} := [x(p,q)] \in \mathbb{R}^{v\times d}$ and task weight vector $\mathbf{w} := [w(q)] \in \mathbb{R}^{d\times 1}$ stored on the master node

1: Quantize $Q(\mathbf{X}) \leftarrow \mathbf{X}$
2: Encode $\mathbf{X}'_j \leftarrow \sum_{i=1}^{k} g_{ij} Q(\mathbf{X}_i),\ j \in [1:n]$
3: Distribute $\mathbf{X}'_j$ and $Q(\mathbf{w})$ to edge device $j \in [1:n]$
4: **for coded training data** $j = 1, 2, \cdots, n$ **in parallel over** n **edge devices (local computations) do**
5: Compute $\mathbf{y}'_j := \mathbf{X}'_j Q(\mathbf{w}) \in \mathbb{R}^{\frac{v}{k}\times 1}$
6: **return** $\mathbf{y}'_j + \mathbf{e}_j$ back to the master node
7: **end for**
8: **for quantization levels** $l = 1, 2, \cdots, N$ **do**
9: Call level-l decoder, returning level-l candidates $\tilde{m}_l(p)$ satisfying the parity check
10: **end for**
11: Form $\hat{\mathbf{r}}'_i(p)$ by the combinations of $\tilde{m}_l(p)$, computing the likelihoods L_i
12: Recover $\hat{\mathbf{r}}(p)$ such that $\hat{\mathbf{r}}(p)G = \hat{\mathbf{r}}'_{i^*}(p)$, where $i^* = \arg\max_i |L_i|$
13: **return** $\hat{\mathbf{y}} \leftarrow \left[\hat{\mathbf{y}}_1^T\ \hat{\mathbf{y}}_2^T\ \cdots \hat{\mathbf{y}}_k^T\right]$, where $\hat{\mathbf{r}}(p) := [\hat{y}_1(p)\ \hat{y}_2(p) \cdots \hat{y}_k(p)]$ for $p \in [1:\frac{v}{k}]$

A. Quantization, Data Encoding and Coded Computations

In this section, we denote the (p,q)-elements of the training data $\mathbf{X}$ by $x(p,q)$ and outline the quantization, encoding, and coded computation steps in lines 1 to 7 of Algorithm 1.

Quantization. The training data $\mathbf{X}$ are quantized with N_1 levels into $Q(\mathbf{X}) := [Q(x(p,q))]$ by the binary representation with remapping terms as follows:

$$Q(x(p,q)) = \gamma_1 \left(\sum_{r=1}^{N_1} 2^{r-1} b_{x(p,q),r} - \beta_1 \right),$$

where the coefficients $b_{x(p,q),r}$ are chosen to minimize the quantization error $|x(p,q) - Q(x(p,q))|$ and γ_1 and β_1 scale the training data $x(p,q)$ so that it falls within the achievable quantization range, obtained by Algorithm 5 in Appendix B.

Denote the $(q,1)$-element of $\mathbf{w}$ by $w(q)$. Similar to the training data, the task weight vector $\mathbf{w}$ is also quantized with N_2 levels into $Q(\mathbf{w}) := [Q(w(q))]$ by the binary representation with remapping terms as follows:

$$Q(w(q)) = \gamma_2 \left(\sum_{r=1}^{N_2} 2^{r-1} b_{w(q),r} - \beta_2 \right),$$

where again the $b_{w(q),r}$'s are chosen to minimize the quantization error $|w(q) - Q(w(q))|$ and γ_2 and β_2 are scaling factors obtained by Algorithm 5.

Data encoding. To encode the quantized training data $Q(\mathbf{X})$, we divide it into k row-splits $\left[Q(\mathbf{X}_1^T)\ Q(\mathbf{X}_2^T)\ \cdots\ Q(\mathbf{X}_k^T)\right]^T \in \mathbb{R}^{v\times d}$, where $Q(\mathbf{X}_i) \in \mathbb{R}^{\frac{v}{k}\times d}$. The quantized data are then encoded as n linear combinations of these k row-splits, i.e., $\mathbf{X}'_j = \sum_{i=1}^{k} g_{ij} Q(\mathbf{X}_i),\ j \in [1:n]$ where the generator matrix $G = [g_{ij}] \in \mathbb{Z}_2^{k\times n}$. We choose n to equal the number of edge devices, so that each device is sent a coded computation, and choose k depending on the estimated straggler probability. Denote the (p,q)-elements of $Q(\mathbf{X}_i)$ and $\mathbf{X}'_j$ by $Q(x_i(p,q))$ and $x'_j(p,q)$, respectively, for $p = [1:\frac{v}{k}]$ and $q = [1:d]$. We can then write the collection of the (p,q)-elements in each of the k row-splits as $Q(\mathbf{x}(p,q)) \equiv [Q(x_1(p,q))\ Q(x_2(p,q))\ \cdots\ Q(x_k(p,q))]$. Similarly, we define $\mathbf{x}'(p,q) \equiv [x'_1(p,q)\ x'_2(p,q)\ \cdots\ x'_n(p,q)]$ as the collection of (p,q)-elements from n coded row-splits $\mathbf{X}'_j$. Thus, the quantization encoding procedure for the (p,q)-element of $\mathbf{X}$ can be summarized as:

$$\mathbf{x}'(p,q) = Q(\mathbf{x}(p,q))\, G.$$

The master node distributes the quantized task weight vector $Q(\mathbf{w})$ and the coded row-splits of the training data, $\mathbf{X}'_j = \left[x'_j(p,q)\right] \in \mathbb{R}^{\frac{v}{k}\times d}$, with coding information $\{g_{ij}\},\ i \in [1:k]$ in its header, to each of its n edge devices.

Coded computations. Each edge device j aims to compute the multiplication of its coded row-split training data with the

quantized task weight vector, i.e., to find $\mathbf{y}'_j := \mathbf{X}'_j Q(\mathbf{w}) \in \mathbb{R}^{\frac{v}{k}\times 1}$. We assume that, instead of sequentially computing each of the $\frac{v}{k}$ elements of $\mathbf{y}'_j$, the device randomly permutes these elements and computes them in random order. Unfinished computations, e.g., if the device does not have time to compute some of the $\mathbf{y}'_j$ elements, would then appear at random locations in $\mathbf{y}'_j$, allowing us to treat incomplete computations as random errors.

At deadline T_{dl}, edge device j returns its (intermediate or complete) coded computation $\mathbf{y}'_j + \mathbf{e}_j$ back to the master node. We assume that the device also sends the locations of unfinished computations $\{p|\mathbf{e}_j(p) \neq 0\}$ and the number of unfinished calculations at each location $\tilde{p}$ (or more precisely, the locations of $Q(w(q))$ corresponding to unfinished calculations at $\tilde{p}$) in its control information. The distribution of the error vector $\mathbf{e}_j$ is estimated by Algorithm 6 in Appendix C based on this information.

The aforementioned procedure of data and task weight vector quantization, quantized data encoding and distribution by the master node and computations of coded data and weight vector on edge devices is summarized by Algorithm 2.

Algorithm 2 Quantization, Data Encoding and Coded Computation

Input: Training data $\mathbf{X} := [x(p,q)] \in \mathbb{R}^{v\times d}$ and task weight vector $\mathbf{w} := [w(q)] \in \mathbb{R}^{d\times 1}$ stored on the master node

1: $Q(\mathbf{X}) \leftarrow [Q(x(p,q))] = \left[\gamma_1 \left(\sum_{r=1}^{N_1} 2^{r-1} b_{x(p,q),r} - \beta_1\right)\right]$ (Algorithm 5)

2: $Q(\mathbf{w}) \leftarrow [Q(w(q))] = \left[\gamma_2 \left(\sum_{r=1}^{N_2} 2^{r-1} b_{w(q),r} - \beta_2\right)\right]$

3: Divide $Q(\mathbf{X})$ into $\left[Q(\mathbf{X}_1^T)\ Q(\mathbf{X}_2^T)\ \cdots\ Q(\mathbf{X}_k^T)\right]^T$, where $Q(\mathbf{X}_i) \in \mathbb{R}^{\frac{v}{k}\times d}$ for $i \in [1:k]$

4: $\mathbf{X}'_j \leftarrow \sum_{i=1}^{k} g_{ij} Q(\mathbf{X}_i),\ j \in [1:n]$ by a generator matrix $G = [g_{ij}] \in \mathbb{Z}_2^{k\times n}$ i.e., $\mathbf{x}'(p,q) \leftarrow Q(\mathbf{x}(p,q))\, G$ for $p \in [1:\frac{v}{k}]$ and $q \in [1:d]$

5: Distribute $\mathbf{X}'_j = \left[x'_j(p,q)\right] \in \mathbb{R}^{\frac{v}{k}\times d}$ with $\{g_{ij},\ i \in [1:k]\}$ to its edge device $j \in [1:n]$

6: Distribute $Q(\mathbf{w})$ to its all edge devices

7: **for coded training data** $j = 1, 2, \cdots, n$ **in parallel over** n **edge devices (local computations) do**

8: Compute $\mathbf{y}'_j \leftarrow \mathbf{X}'_j Q(\mathbf{w}) \in \mathbb{R}^{\frac{v}{k}\times 1}$ by random permutation order

9: **return** $\mathbf{y}'_j + \mathbf{e}_j$ back to the master node with location information $\{p|\mathbf{e}_j(p) \neq 0\}$ and the number of unfinished calculations at each location $\tilde{p} \in \{p|\mathbf{e}_j(p) \neq 0\}$ (or more precisely, the locations of $Q(w(q))$ corresponding to unfinished calculations at $\tilde{p} \in \{p|\mathbf{e}_j(p) \neq 0\}$)

10: **end for**

B. Computation Decoding

The master node finally decodes all of the received coded computations $\{\mathbf{y}'_j + \mathbf{e}_j \mid j \in [1:n]\}$ to recover the actual (quantized) computations corresponding to each of the k row-splits in the training data, i.e., $\{\mathbf{y}_i \mid \mathbf{y}_i := Q(\mathbf{X}_i)\, Q(\mathbf{w}) \in \mathbb{R}^{\frac{v}{k}\times 1},\ i \in [1:k]\}$. Denote the $(p,1)$-elements of $\mathbf{y}_i$ and $\mathbf{y}'_j$ by $y_i(p)$ and $y'_j(p)$, respectively. We use $\mathbf{r}(p)$ to denote the collection of the $(p,1)$-elements $[y_1(p)\ y_2(p)\ \cdots\ y_k(p)]$ from the k original computations $\mathbf{y}_i$, which we are trying to compute. We analogously use $\mathbf{r}'(p) \equiv [y'_1(p)\ y'_2(p)\ \cdots\ y'_n(p)]$ to denote the collection of the $(p,1)$-elements from the n coded computations $\mathbf{y}'_j = \mathbf{X}'_j Q(\mathbf{w})$, and $\mathbf{z}(p) \equiv [e_1(p)\ e_2(p)\ \cdots\ e_n(p)]$ to denote the collection of $(p,1)$-elements in the n errors $\mathbf{e}_j$.

We show that a generator matrix G for encoding original data into coded data can also be used for encoding the actual computations into coded computations.

Theorem 1. *If the encoding procedure for the original data* $\mathbf{X}$ *is expressed in terms of each* (p,q)*-element as follows:*

$$\mathbf{x}'(p,q) = Q(\mathbf{x}(p,q))G \tag{1}$$

and the original and coded computations are given by $\mathbf{y}_i = Q(\mathbf{X}_i)Q(\mathbf{w})$ *for* $i = [1:k]$ *and* $\mathbf{y}'_j = \mathbf{X}'_j Q(\mathbf{w})$ *for* $j = [1:n]$*, respectively, the encoding procedure for the original computations is also expressed using the same generator matrix* G *in terms of* $(p,1)$*-element as follows:*

$$\left[\mathbf{m}'_{b,l}(p) - \sum_{r=1}^{l-1} 2^{r-l}\left(\mathbf{m}_{b,r}(p)G - \mathbf{m}'_{b,r}(p)\right)\right] \mod 2$$
$$= \mathbf{m}_{b,l}(p)G \mod 2,$$

where $\mathbf{m}_{b,r}(p) = [b_{r,1}(p)\ b_{r,2}(p)\ \cdots\ b_{r,k}(p)]$, $\mathbf{m}'_{b,r}(p) = [b'_{r,1}(p)\ b'_{r,2}(p)\ \cdots\ b'_{r,n}(p)]$,

$$\mathbf{r}(p) = \gamma_1\gamma_2\left(\sum_{r=1}^{N} 2^{r-1}\left[b_{r,1}(p)\ \cdots\ b_{r,k}(p)\right] - \beta'_p \mathbf{1}_{1\times k}\right),$$

and

$$\mathbf{r}'(p) = \gamma_1\gamma_2\left(\sum_{r=1}^{N'} 2^{r-1}\left[b'_{r,1}(p)\ \cdots\ b'_{r,n}(p)\right] - \beta'_p \mathbf{1}_{1\times k} G\right).$$

Proof. See the proof in Appendix A. □

The decoding procedure then proceeds sequentially across the quantization levels, and finally the results from each level are combined to recover the actual computation $\hat{\mathbf{y}}$, our estimate of $\mathbf{y} = \mathbf{X}\mathbf{w}$.

Choosing candidates with sequential decoding. For each $p = [1:\frac{v}{k}]$, the master node employs sequential decoding across quantization levels; it computes a candidate set of coded bits $\hat{\mathbf{m}}_{j,l}(p)$ for each quantization level of coded computations $\mathbf{r}'(p)$. We describe this procedure for recovering the level-1 coded computations; subsequent levels are decoded analogously. Applying the remapping operation to

$$\mathbf{u}(p) = \mathbf{r}'(p) + \mathbf{z}(p) = \mathbf{r}(p)G + \mathbf{z}(p),$$

we write the level-1 received coded computation as

$$\tilde{\mathbf{u}}_{\text{level},1}(p) = \frac{1}{\gamma_1\gamma_2}\mathbf{u}(p) + \beta'_p \mathbf{1}_{1\times k} G,$$
$$= \sum_{r=1}^{N} 2^{r-1}\mathbf{m}'_{b,r}(p) + \mathbf{z}_{\text{level},1}(p), \tag{2}$$

where $\mathbf{m}'_{b,r}(p) := \left[b'_{r,1}(p)\ b'_{r,2}(p)\ \cdots\ b'_{r,n}(p)\right]$ consists of the coded quantization bits for level r in $\mathbf{r}'(p)$, $N = N_1+N_2+1$, $\mathbf{z}_{\text{level},1}(p) = \frac{1}{\gamma_1\gamma_2}\mathbf{z}(p)$ and

$$\beta'_p = -\frac{\min\left([Q(\mathbf{x}^T(p,1))\ \cdots\ Q(\mathbf{x}^T(p,d))]Q(\mathbf{w})\right)}{\gamma_1\gamma_2}.$$

We can estimate β'_p, the minimum of the quantized computations rescaled by $\gamma_1\gamma_2$, from the recovered computations in the previous epoch of data analytics algorithms, task weight vector $\mathbf{w}$ in the current epoch and its differential update. Candidates of n bits in each level can be separately decoded by the modulo-2 operation. Applying the modulo-2 operation on the level-1 received computations $\tilde{\mathbf{u}}_{\text{level},1}(p)$ in (2), all the quantization bits except level-1 bits are completely removed. Hence, the master node sees a noisy version of the level-1 coded computation over $\mathbb{Z}_2^n$.

Given $\tilde{\mathbf{u}}_{\text{level},1}(p)$ and the information on unfinished computations in the header, the master node can identify τ unreliable locations of $\mathbf{m}'_{b,1}(p)$ (i.e., locations where we expect incompleteness errors). It then assumes that these elements are in locations $1 \le j_1 < j_2 < \cdots < j_\tau \le n$ of $\mathbf{m}'_{b,1}(p)$. Denote $\tilde{\mathbf{m}}_1(p)$ by the vector formed by fixing the elements as they are in all other locations except $j_1 < j_2 < \cdots < j_\tau$ of $\mathbf{m}'_{b,1}(p)$. The master node considers all 2^τ level-1 vectors $\tilde{\mathbf{m}}_{1,1}(p),\ \tilde{\mathbf{m}}_{2,1}(p),\ \cdots, \tilde{\mathbf{m}}_{2^\tau,1}(p)$ formed by replacing the elements of $\tilde{\mathbf{m}}_1(p)$ in locations $j_1 < j_2 < \cdots < j_\tau$ with all possible binary numbers of length τ, i.e., binary expansion of $0,\ 1,\ \cdots, 2^\tau - 1$. It then identifies level-1 candidates $\tilde{\mathbf{m}}_{j,1}(p),\ j \in [1 : 2^\tau]$ satisfying $\tilde{\mathbf{m}}_{j,1}(p)H^T = \mathbf{0}_{n-k}$, i.e., codewords of G. Without loss of generality, assume that the level-1 candidates are $\hat{\mathbf{m}}_{1,1}(p),\ \hat{\mathbf{m}}_{2,1}(p),\ \cdots,\ \hat{\mathbf{m}}_{c_1,1}(p)$ for some $c_1 \in [1 : 2^\tau]$. Note that $c_1 = 0$ if none of $\tilde{\mathbf{m}}_{j,1}(p)$ satisfies $\tilde{\mathbf{m}}_{j,1}(p)H^T = \mathbf{0}$. For each $\hat{\mathbf{m}}_{j,1}(p),\ j \in [1 : c_1]$, if $c_1 > 0$, using the level-1 received computation $\tilde{\mathbf{u}}_{\text{level},1}(p)$ and the distribution of $\frac{1}{\gamma_1\gamma_2}\mathbf{z}(p)$ based on Algorithm 6, the master node computes the log-likelihood of $\hat{\mathbf{m}}_{j,1}(p),\ j \in [1 : c_1]$. It then chooses the $\hat{\mathbf{m}}_{j,1}(p)$ with the highest absolute value of log-likelihood as potentially recovered level-1 coded bits denoted by $\hat{\mathbf{m}}_{b,1}(p)$ and keeps all the level-1 candidates $\hat{\mathbf{m}}_{j,1}(p),\ j \in [1 : c_1]$ for the final stage of decoding. If $c_1 = 0$, the master node sets the potentially recovered level-1 coded bits by randomly choosing 0 or 1 in locations $j_1,\ j_2,\ \cdots, j_\tau$.

The master then cancels out the contribution of potentiually recovered level-1 coded bits $\hat{\mathbf{m}}_{b,1}(p)$ from $\tilde{\mathbf{u}}_{\text{level},1}(p),\ p \in [1 : \frac{n}{k}]$ and scales by $\frac{1}{2}$ to get

$$\tilde{\mathbf{u}}_{\text{level},2}(p) = \sum_{r=2}^{N} 2^{r-2}\mathbf{m}'_{b,r}(p) + \frac{1}{2}\left(\mathbf{m}'_{b,1}(p) - \hat{\mathbf{m}}_{b,1,1:k}(p)G\right) + \mathbf{z}_{\text{level},2}(p),$$

where $\mathbf{z}_{\text{level},2}(p) = \frac{1}{2\gamma_1\gamma_2}\mathbf{z}(p)$. It then can attempt to decode for level-2 coded computation by following the same steps as in the decoding operation of level-1 coded computation. The recursive procedure continues until coded computation candidates up to level N are decoded.

Combining the candidate results. Now the master node chooses N_τ so that the size of the candidates set $\prod_{l=N_\tau+1}^{N} c_l \le C$, where C is the predetermined maximum value. It then considers all possible combinations of candidates from level-1 to level-N by fixing a level-l candidate as potentially recovered level-l coded bits $\hat{\mathbf{m}}_{b,l}(p)$ for $l \in [1 : N_\tau]$ and recovers the corresponding coded computations $\hat{\mathbf{r}}'_i(p),\ i \in [1 : \prod_{l=N_\tau+1}^{N} c_l]$, because higher quantization bits are more important than lower quantization bits in decoding when reducing the size of candidates set. Using the received coded computation $\mathbf{u}(p)$ and the distribution of $\mathbf{z}(p)$ based on Algorithm 6, it computes the log-likelihood of each combination and identifies the recovered coded computation $\hat{\mathbf{r}}'(p)$ as the one with the highest absolute values of log-likelihood. The master node then recovers the desired actual computation $\hat{\mathbf{r}}(p)$ that, when encoded with the generator matrix G, produces the coded compuation $\hat{\mathbf{r}}'(p)$. Combining $\hat{\mathbf{r}}(p) = [\hat{y}_1(p)\ \hat{y}_2(p)\cdots\hat{y}_k(p)]$ for $p \in \left[1 : \frac{v}{k}\right]$, it recovers the actual computation $\hat{\mathbf{y}} = \left[\hat{\mathbf{y}}_1^T\ \hat{\mathbf{y}}_2^T\ \cdots\hat{\mathbf{y}}_k^T\right]$. The proposed decoding procedure is summarized in Algorithm 3.

Algorithm 3 Computation Decoding

for computation elements $p = 1, 2, \cdots, \frac{v}{k}$ **in parallel do**
  $\mathbf{r}'(p) \leftarrow [y'_1(p)\ y'_2(p)\ \cdots\ y'_n(p)]$
  $\mathbf{z}(p) \leftarrow [e_1(p)\ e_2(p)\ \cdots\ e_n(p)]$
  $\mathbf{u}(p) \leftarrow \mathbf{r}'(p) + \mathbf{z}(p)$
  $\beta'_p = -\frac{\min\left([Q(\mathbf{x}^T(p,1))\ \cdots\ Q(\mathbf{x}^T(p,d))]Q(\mathbf{w})\right)}{\gamma_1\gamma_2}$
  $N \leftarrow N_1 + N_2 + 1$
  for quantization levels $l = 1, 2, \cdots, N$ **do**
    if $l = 1$ **then**
      $\tilde{\mathbf{u}}_{\text{level},1}(p) \leftarrow \frac{1}{\gamma_1\gamma_2}\mathbf{u}(p) + \beta'_p\mathbf{1}_{1\times k}G = \sum_{r=1}^{N} 2^{r-1}\mathbf{m}'_{b,r}(p) + \mathbf{z}_{\text{level},1}(p)$, where $\mathbf{m}'_{b,r}(p) := \left[b'_{r,1}(p)\ b'_{r,2}(p)\ \cdots\ b'_{r,n}(p)\right]$ and $\mathbf{z}_{\text{level},1}(p) := \frac{1}{\gamma_1\gamma_2}\mathbf{z}(p)$
    else if $l \ge 2$ **then**
      $\tilde{\mathbf{u}}_{\text{level},l}(p) \leftarrow \frac{1}{2}\left(\tilde{\mathbf{u}}_{\text{level},l-1}(p) - \hat{\mathbf{m}}_{b,l-1}(p)\right) = \sum_{r=l}^{N} 2^{r-l}\mathbf{m}'_{b,r}(p) + \sum_{r=1}^{l-1} 2^{r-l}\left(\mathbf{m}'_{b,r}(p) - \hat{\mathbf{m}}_{b,r,1:k}(p)G\right) + \mathbf{z}_{\text{level},l}(p)$, where $\mathbf{z}_{\text{level},l}(p) := \frac{1}{2^{l-1}\gamma_1\gamma_2}\mathbf{z}(p)$.
    end if
    $\tilde{\mathbf{u}}_{\text{level},l}(p) \mod 2 \leftarrow \Delta_l(p) + (\mathbf{z}_{\text{level},l}(p) \mod 2)$, where $\Delta_l(p) := \left(\mathbf{m}'_{b,l}(p) + \sum_{r=1}^{l-1} 2^{r-l}\left(\mathbf{m}'_{b,r}(p) - \hat{\mathbf{m}}_{b,r,1:k}(p)G\right)\right) \mod 2$
    Find unreliable locations $1 \le j_1 < j_2 < \cdots < j_\tau \le n$ of $\Delta_l(p)$ from the information on unfinished computations
    Form $\tilde{\mathbf{m}}_l(p)$ by fixing the values of the elements of $\Delta_l(p)$ in all other locations except $j_1, j_2, \cdots, j_\tau$ of $\Delta_l(p)$
    Form $\tilde{\mathbf{m}}_{j,l}(p), j \in [1 : 2^\tau]$ by replacing the elements of $\tilde{\mathbf{m}}_l(p)$ in locations $j_1, j_2, \cdots, j_\tau$ with binary expansion of $[0 : 2^\tau - 1]$

Find $\{\tilde{\mathbf{m}}_{j,l}(p)\}$ satisfying $\tilde{\mathbf{m}}_{j,l}(p)H^T = \mathbf{0}_{n-k}$
if $|\{\tilde{\mathbf{m}}_{j,l}(p)|\tilde{\mathbf{m}}_{j,l}(p)H^T = \mathbf{0}_{n-k}\}| := c_l \geq 1$ **then**
Denote such level-1 candidates set by $\{\hat{\mathbf{m}}_{j,l}(p),\ j \in [1 : c_l]\}$
Compute the log-likelihood $L_{j,l}$ of $\hat{\mathbf{m}}_{j,l}(p)$ for $j \in [1 : c_l]$ using Algorithm 6
$\hat{\mathbf{m}}_{b,l}(p) \leftarrow \hat{\mathbf{m}}_{j^*,l}(p)$, where $j^* = \arg\max_{j\in[1:c_l]} |L_{j,l}|$
else
$\hat{\mathbf{m}}_{b,l}(p) \leftarrow \tilde{\mathbf{m}}_{j,l}(p)$ by randomly choosing $j \in [1 : 2^\tau]$
end if
end for
Choose N_τ satisfying $\prod_{l=N_\tau+1}^{N} c_l \leq C$, where C is the maximum size of the candidates set
Recover $\hat{\mathbf{r}}'_i(p), i \in [1 : \prod_{l=N_\tau+1}^{N} c_l]$ from all possible combinations of level-1 to level-N candidates by fixing a level-l candidate as $\hat{\mathbf{m}}_{b,l}(p)$ for $l \in [1 : N_\tau]$
Compute L_i of $\hat{\mathbf{r}}'_i(p)$ using Algorithm 6
$\hat{\mathbf{r}}'(p) \leftarrow \hat{\mathbf{r}}'_{i^*}(p)$, where $i^* = \arg\max_i |L_i|$
Recover $\hat{\mathbf{r}}(p)$ such that $\hat{\mathbf{r}}'(p) = \hat{\mathbf{r}}(p)G$
end for
return $\hat{\mathbf{y}} \leftarrow \left[\hat{\mathbf{y}}_1^T\ \hat{\mathbf{y}}_2^T\ \cdots \hat{\mathbf{y}}_k^T\right]$, where $\hat{\mathbf{r}}(p) = [\hat{y}_1(p)\ \hat{y}_2(p) \cdots \hat{y}_k(p)]$ for $p \in [1 : \frac{v}{k}]$

C. Data Exchange

In order to implement coded edge computing algorithms in a practical system, the master node and edge devices must exchange both data and control information. We outline these exchanges, as well as state information required by the master node, in this section. In particular, edge devices that help with multiple rounds of computation can cache coded training data from prior rounds to improve the performance of the proposed coded computing strategy.

Initial state. The master node may not initially know the full network topology due to the distributed, mobile and ad-hoc properties of edge computing environments. Thus, the master node must first use a discovery process to find a set of edge devices available to help carry out its computations.

After this discovery process, the master node sends **data and control information to each device** j. The data consists of the coded row-split training data $\mathbf{X}'_j$ and the quantized, current task weight vector $Q(\mathbf{w})$. The control information includes two commands: a command to multiply each row of $\mathbf{X}'_j$ by $Q(\mathbf{w})$ after randomly permuting the row order, and a command to return the (possibly unfinished) coded computations, with the indices of the incomplete rows, at the deadline T_{dl}.

In epoch ≥ 2, unlike in epoch 1, the master node can take advantage of cached coded training data at edge devices from previous epochs and send the new coded training data via coded multicast to all the previously joined edge devices and newly joined edge devices. Since new devices do not have cached data, it also sends (different) newly coded data via unicast to these nodes.

This data information comes with **control information**, which includes coding information on how the coding on the newly coded data was carried out. If the edge device j has cached coded data, the master node sends a coded shuffling command to carry out coding based on both the newly coded data and cached coded data to generate another set of newly coded training data that can be stored for later computations. If not, the edge device j simply stores the sent data and uses it for computations.

If edge device acts as a relay node and has cached coded data, the master node conveys a command to carry out encoding the newly coded data and cached coded data into the other newly coded data and to forward it to another edge device, if the edge device has cached coded data. In the same way as in epoch 1, it also conveys a command to carry out the multiplication of aforementioned new coded data and the current task weight vector with random permutations and a command to return complete coded computations within deadline.

Each edge device j that is still active at the deadline T_{dl} then sends the master node **data information** consisting of the (intermediate or complete) coded computations and error information on the number of unfinished calculations per computation element.

State for decoding. The master node reconstructs the generator matrix G, which depends on received coded computations. Some coded computations might not be returned due to mobility, battery, or network congestion.

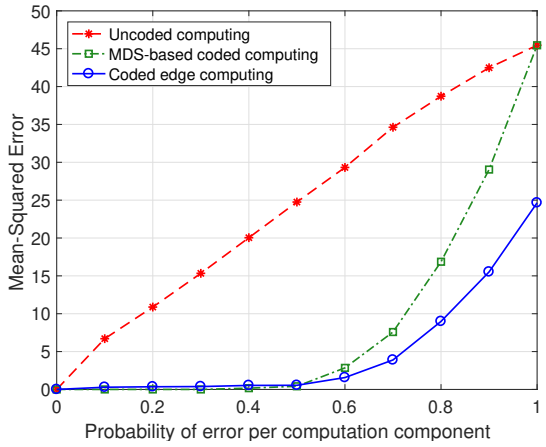
IV. Simulation Results

We demonstrate coded edge computing's advantages over MDS-based coded computing for matrix-vector multiplication both alone and within federated multi-task learning.

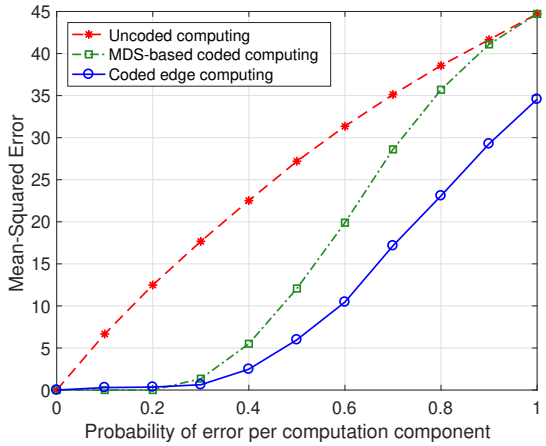
A. Coded Edge Computing for Matrix-Vector Multiplication

To evaluate the performance of coded edge computing in practice, we consider a simulation with a master node and $n = 15$ edge devices and compare our coded edge computing strategy to MDS-based coded computing and uncoded computing. We assume 7 non-stragglers and 8 stragglers among the edge devices. We consider a basic matrix operation $\mathbf{y} = \mathbf{X}\mathbf{w}$, where $\mathbf{X} \in \mathbb{R}^{10k\times 9}$ is synthetic training data sampled from the standard normal distribution and $\mathbf{w}$ is a task weight vector with elements $\in \{-1, 1\}$. We evaluate the error of the recovered computation when we vary k, the number of row splits, as well as the probability a straggler cannot finish its computations and the level of quantization.

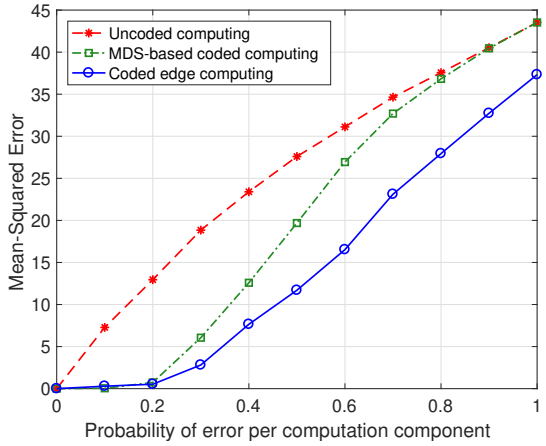
The training data $\mathbf{X}$ and task weight vector $\mathbf{w}$ are respectively quantized with $N_1 = 8$ and $N_2 = 2$ bits. To test the effect of k, the number of row splits, we run three experiments where we divide the quantized training data into $k = 8, 10, 11$ sub-data splits. In each experiment, the sub-data are then encoded into 15 coded data elements to be distributed among the $n = 15$ edge devices. Note that, depending on the realized actual training data, the distribution of the coded training data becomes t location-scale distributions (not normal) with



(a) $(n, k) = (15, 8)$.



(b) $(n, k) = (15, 10)$.



(c) $(n, k) = (15, 11)$.

Fig. 2. Performance of coded edge computing with different code rates and 11 bits quantization.

different parameters that determine the computation error distribution. We evaluate the effect of device stragglers by running 11 experiments where we sweep the probability each straggler cannot finish computing $y'(p)$ from $\{0, 0.1, \cdots, 1\}$; each straggler's ability to finish is independent from the others'. Nodes that do not finish within a given deadline

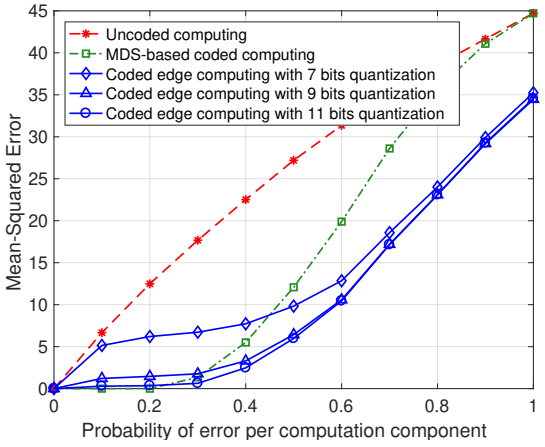


Fig. 3. Performance of (15,10)-coded edge computing with different quantization bits.

return partially finished computations. When τ errors occur, we assume that each error consist of $s \in [1 : \tau]$ unfinished calculations, and that the number of unfinished calculations per element is conveyed to the master node from edge device. The computation decoder of coded edge computing is simulated with 11 levels of multi-stage decoding per collection of 15 coded computations elements. We assume that MDS-based coded computing performs decoding per collection of $(p, 1)$-elements from the 15 coded computation elements, i.e., it can recover the actual computation based on each collection instead of coded subdata.

Simulation results for the mean-squared error (MSE, defined as the 2-norm of the difference between the actual computation and the recovered computation) demonstrate that coded edge computing outperforms MDS-based coded computing and uncoded computing uniformly for all error probabilities and numbers of row splits k, as shown in Figure 2. For example, (15,10)-coded edge computing outperforms MDS-based coded computing by 21.5–55.3% for error probabilities greater than or equal to 0.3. In particular, coded edge computing achieves a larger performance gap from the achievable MSE of MDS-based coded computing for a lower code rate, because it can handle stragglers by exploiting partial coded computations with larger parity information more efficiently than MDS-based coded computing. As we would expect, the MSE for all methods increases with the error probability (since more errors will be made) and with k (since more computation results are needed to recover the original computation), but coded edge computing consistently exhibits a lower rate of increase.

Finally, Figure 3's simulation results for MSE demonstrate that reducing the number of quantization bits from 11 to 7 degrades the performance of coded edge computing. For example, coded edge computing can recover quantized actual computations almost perfectly for lower error probabilities less than or equal to 0.3, but suffers from inherent quantization errors. Thus, the selection of quantization bits for coded edge computing can be chosen to maximize performance within the predetermined range depending on data analytics algorithms,

while reducing the number of decoding stages.

B. Coded Edge Computing for Federated Multi-Task Learning

We finally apply our coded edge computing strategy to federated learning as a case study. Federated multi-task learning (FMTL) has been proposed to overcome the system and statistical challenges in the federated setting [18]. FMTL aims to learn local models for each device, namely FMTL node, from data stored locally at the device, by solving the following empirical risk minimization (ERM) in the federated setting:

$$\min_{\mathbf{W},\mathbf{\Omega}} \sum_{t=1}^{m} \sum_{i=1}^{n_t} l_t(\mathbf{w}_t, \mathbf{x}_t^i) + \mathcal{R}(\mathbf{W}, \mathbf{\Omega}),$$

where $\mathbf{W}$, $\mathbf{\Omega}$, $l_t(\mathbf{w}_t, \mathbf{x}_t^i)$, and $\mathcal{R}(\mathbf{W}, \mathbf{\Omega})$ are models, task relationships, losses, and a regularizer, respectively. In an alternating fashion, a FMTL node learns a local model by returning an approximate solution on each device based on a fixed task relationship matrix and local data, and sending the update $\mathbf{v}_t$ to the central server.

We take into account the edge environment by supposing there are a multitude of edge devices that can communicate locally with one of the FMTL nodes. We adapt the proposed method for federated multi-task learning, as shown in Algorithm 4. Our coded strategy is applied to the multiplication of local training data with a task weight vector, which is executed when the local solver is called at a FMTL node to return approximate solutions back to the central server from the node.

Algorithm 4 FMTL [18] with the proposed coded strategy

Input: Data $\mathbf{X}_t$ stored on $t = 1, \ldots, m$ devices
Initialize $\boldsymbol{\alpha}^{(0)} := \mathbf{0}$, $\mathbf{v}^{(0)} := \mathbf{0}$
for iterations $i = 0, 1, \ldots$ **do**
  for iterations $h = 0, 1, \ldots, H_i$ **do**
    for devices $t \in \{1, 2, \ldots, m\}$ **in parallel do**
      call local solver, returning θ_t^h-approximate solution $\Delta\boldsymbol{\alpha}_t$ by carrying out coded edge computing in FMTL node t and its edge devices
      update local variables $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + \Delta\boldsymbol{\alpha}_t$
    end for
    update $\mathbf{v} \leftarrow \mathbf{v} + \sum_t \mathbf{X}_t \Delta\boldsymbol{\alpha}_t$
  end for
  Update Ω centrally using $\mathbf{w}(\mathbf{v}) := \nabla\mathcal{R}^*(\mathbf{v})$
end for
$\mathbf{w}(\mathbf{v}) := \nabla\mathcal{R}^*(\mathbf{v})$
return $\mathbf{W} := [\mathbf{w}_1, \cdots, \mathbf{w}_m]$

To evaluate the performance of coded edge computing in FMTL, we consider a simulation with $m = 17$ tasks (or FMTL nodes), each of which has $n = 15$ edge devices to which it connects, throughout the training. An edge device carries out coded computing for $H_i = 1300$ rounds of computation. We compare our coded edge computing strategy to MDS-based coded computing and uncoded computing. As in Section IV-A, we assume 7 non-stragglers and 8 stragglers among the edge devices. We consider the same synthetic training data as in Section IV-A and training and test output vectors from FMTL's small dataset [18]. The training data $\mathbf{X}$ and task weight vector $\mathbf{w}$ are both quantized with 5 bits. The number of row splits is $k = 10$. Control information overhead of each edge device is 16 bits. We fix the probability that each straggler cannot finish computing each element in its output vector as 0.5; again each straggler's ability to finish is independent from the others'. We estimate wall-clock time (in milliseconds) to run federated learning as in [18].

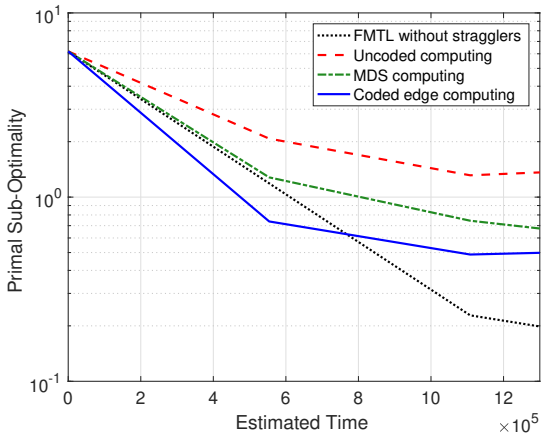


Fig. 4. Performance of FMTL with coded edge computing, MDS-based coded computing and uncoded computing. $(n, k) = (15, 10)$. Eight edge devices among 15 become stragglers with the probability of 0.5.

Simulation results for primal suboptimality (defined by the difference between optimal ERM value and ERM values of each scheme) demonstrate that coded edge computing outperforms MDS-based coded computing and uncoded computing uniformly for all estimated time. For example, coded edge computing outperforms MDS-based coded computing and uncoded computing by 18.35% and 69.5% for estimated time 800000, respectively. We note that, due to quantization effects, FMTL with coded edge computing seems to outperform FMTL without any stragglers in the beginning of training time, but as learning progresses the quantization error becomes more prominent.

V. Concluding Remarks

In this work, we propose a coded computing strategy for dynamically changing edge environments, which pose new heterogeneity and reliability challenges for executing distributed computing tasks. In particular, in distributed computing settings we cannot rely on edge devices to finish their computations within a given deadline (if at all). Our key methodological insights are to carefully combine data encoding and computation decoding via quantization and sphere/multi-stage decoding via modulo operations, which allow us to exploit partially finished computations from edge devices. Thus, our scheme is robust to failures (e.g., from power failures or interruptions in network connectivity) and long runtimes at edge devices, unlike conventional coded computing schemes. Performance simulations show that our proposed approach offers significant gains in both matrix-

vector multiplications and its application to federated learning over conventional schemes.

ACKNOWLEDGEMENT

This work was supported in part by NSF Grant CNS-1909306 and the Defense Advanced Research Projects Agency (DARPA) under contract no. HR001117C0052. The views expressed are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

APPENDIX A
PROOF OF THEOREM 1

We first want to show that $\mathbf{r}'(p) = \mathbf{r}(p)G$. The coded computations of $(p,1)$-elements $\mathbf{r}'(p)$ are given by

$$\begin{aligned}\mathbf{r}'(p) &= [y'_1(p) \ \cdots \ y'_n(p)] \\ &= \left[\sum_{q=1}^{d} x'_1(p,q)Q(w(q)) \ \cdots \ \sum_{q=1}^{d} x'_n(p,q)Q(w(q))\right] \\ &= \sum_{q=1}^{d} Q(w(q))\mathbf{x}'(p,q) \\ &= \sum_{q=1}^{d} Q(w(q))Q(\mathbf{x}(p,q))G \\ &= \left[\sum_{q=1}^{d} Q(x_1(p,q))Q(w(q)) \cdots \sum_{q=1}^{d} Q(x_k(p,q))Q(w(q))\right] G \\ &= [y_1(p) \ \cdots \ y_k(p)]G \\ &= \mathbf{r}(p)G.\end{aligned}$$

By remapping both sides,

$$\begin{aligned}\frac{\mathbf{r}'(p)}{\gamma_1\gamma_2} + \beta'_p \mathbf{1}_{1\times k}G &= \frac{\mathbf{r}(p)G}{\gamma_1\gamma_2} + \beta'_p \mathbf{1}_{1\times k}G \\ \sum_{r=1}^{N'} 2^{r-1}\left[b'_{r,1}(p)\cdots b'_{r,n}(p)\right] &= \sum_{r=1}^{N} 2^{r-1}\left[b_{r,1}(p)\cdots b_{r,k}(p)\right]G.\end{aligned} \tag{3}$$

By taking modulo 2 on both sides, we have

$$\left[b'_{1,1}(p) \ \cdots \ b'_{1,n}(p)\right] = [b_{1,1}(p) \ \cdots \ b_{1,k}(p)]\,G \mod 2. \tag{4}$$

We then cancel out the contribution of the level-1 coded bits in (4) from (3) as follows.

$$\begin{aligned}&\frac{1}{2}\left(\sum_{r=1}^{N'} 2^{r-1}\left[b'_{r,1}(p)\cdots b'_{r,n}(p)\right] - \left[b_{1,1}(p)\cdots b_{1,k}(p)\right]G\right) \\ &= \sum_{r=2}^{N'} 2^{r-2}\left[b'_{r,1}(p) \ \cdots \ b'_{r,n}(p)\right] \\ &\quad + \frac{1}{2}\left(\left[b'_{1,1}(p) \ \cdots \ b'_{1,n}(p)\right] - [b_{1,1}(p) \ \cdots \ b_{1,k}(p)]\,G\right) \\ &= \sum_{r=2}^{N} 2^{r-2}[b_{r,1}(p) \ \cdots \ b_{r,k}(p)]\,G.\end{aligned}$$

By taking modulo 2 on both sides, we have

$$\begin{aligned}&\left\{\left[b'_{2,1}(p) \ \cdots \ b'_{2,n}(p)\right] + \frac{1}{2}\left(\left[b'_{1,1}(p) \ \cdots \ b'_{1,n}(p)\right]\right.\right. \\ &\left.\left. - [b_{1,1}(p) \ \cdots \ b_{1,k}(p)]\,G\right)\right\} \mod 2 \\ &= [b_{2,1}(p) \ \cdots \ b_{2,k}(p)]\,G \mod 2.\end{aligned}$$

By the recursive procedure up to level l, we have

$$\begin{aligned}&\left\{\left[b'_{l,1}(p) \ \cdots \ b'_{l,n}(p)\right] + \sum_{r=1}^{l-1} 2^{r-l}\left(\left[b'_{r,1}(p) \ \cdots \ b'_{r,n}(p)\right]\right.\right. \\ &\left.\left. - [b_{r,1}(p) \ \cdots \ b_{r,k}(p)]\,G\right)\right\} \mod 2 \\ &= [b_{l,1}(p) \ \cdots \ b_{l,k}(p)]\,G \mod 2\end{aligned}$$

for $l = [2:N]$, which completes the proof.

APPENDIX B
QUANTIZATION BY THE BINARY REPRESENTATION

Algorithm 5 Quantization by the Binary Representation

Input: Training data $\mathbf{X} := [x(p,q)] \in \mathbb{R}^{v\times d}$ stored on the master node
Solve $\frac{\min_{p,q} x(p,q)}{\gamma_1} = -\beta_1$ and $\frac{\min_{p,q} x(p,q)}{\gamma_1} + \beta_1 = 2^{N_1} - 1$ for γ_1 and β_1

if $\min_{p,q} x(p,q) < \max_{p,q} x(p,q) \le 0$ **then**
  $\beta_1 \leftarrow \lfloor \beta_1 \rfloor$
  $\gamma_1 \leftarrow \frac{\min_{p,q} x(p,q)}{-\beta_1}$
else if $\min_{p,q} x(p,q) < 0 < \max_{p,q} x(p,q)$ **then**
  Set β_1 as the closest integer among $\lfloor \beta_1 \rfloor$ and $\lceil \beta_1 \rceil$
  $\gamma_1 \leftarrow \max\left(\frac{\min_{p,q} x(p,q)}{-\beta_1}, \frac{\max_{p,q} x(p,q)}{2^{N_1}-1-\beta_1}\right)$
else if $0 \le \min_{p,q} x(p,q) < \max_{p,q} x(p,q)$ **then**
  $\beta_1 \leftarrow \lceil \beta_1 \rceil$
  $\gamma_1 \leftarrow \frac{\max_{p,q} x(p,q)}{2^{N_1}-1-\beta_1}$
end if

APPENDIX C
ESTIMATION OF COMPUTATION ERROR DISTRIBUTION

Algorithm 6 Estimation of Computation Error Distribution

Input: Coded training data $\mathbf{X}' := [x'(p,q)] \in \mathbb{R}^{\frac{vn}{k}\times d}$ stored on the master node
Find the best distribution $f_{X'}(x')$ that fits coded training data $\mathbf{X}'$
for $j = 1, 2, \cdots, n$ **in parallel do**
  for $p \in \{p | \mathbf{e}_j(p) \ne 0\}$ **in parallel do**
    if Locations $k_1, \ k_2, \ \cdots, \ k_{\varphi(j,p)}$ of unfinished calculations are known **then**
      $f_{E_j(p,q)}(x) \leftarrow \frac{1}{|Q(w(q))|} f_{X'}\left(\frac{x}{-Q(w(q))}\right)$
    else if The number $\varphi(j,p)$ of locations of unfinished calculations is known **then**
      $f_{E_j(p,q)}(x) \leftarrow \frac{d}{||Q(\mathbf{w})||_2} f_{X'}\left(\frac{dx}{-||Q(\mathbf{w})||_2}\right), \ q \in \{k_1, \ k_2, \ \cdots, \ k_{\varphi(j,p)}\}$
    end if
    return $f_{E_j(p)}(x) \leftarrow \left(f_{E_j(p,k_1)} * f_{E_j(p,k_2)} * \cdots * f_{E_j(p,k_{\varphi(j,p)})}\right)(x)$
  end for
end for

REFERENCES

[1] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, March 2018.

[2] R. K. Maity, A. S. Rawat, and A. Mazumdar, "Robust gradient descent via moment encoding with LDPC codes," in *Proc. SysML Conference*, Stanford, CA, Feb. 2018.

[3] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded mapreduce," in *Proc. 53rd Ann. Allerton Conf. Comm. Control Comput.*, Sept 2015, pp. 964–971.

[4] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *Proc. 54th Ann. Allerton Conf. Comm. Control Comput.*, Sep. 2016, pp. 954–960.

[5] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proc. of the 34th Int. Conf. on Mach. Learning*, Sydney, Australia, Aug 2017, pp. 3368–3376.

[6] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Coded computation over heterogeneous clusters," *IEEE Trans. Inf. Theory*, vol. 65, no. 7, pp. 4227–4242, July 2019.

[7] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Communication-aware computing for edge processing," in *Proc. IEEE Int. Symp. Inf. Theory*, June 2017, pp. 2885–2889.

[8] C. Karakus, Y. Sun, and S. Diggavi, "Encoded distributed optimization," in *Proc. IEEE Int. Symp. Inf. Theory*, June 2017, pp. 2890–2894.

[9] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Advances in Neural Inf. Process. Syst. 29*. Curran Associates, Inc., 2016, pp. 2100–2108.

[10] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in *Proc. IEEE Int. Symp. Inf. Theory*, June 2017, pp. 2403–2407.

[11] K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Coded computation for multicore setups," in *Proc. IEEE Int. Symp. Inf. Theory*, June 2017, pp. 2413–2417.

[12] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory*, June 2017, pp. 2418–2422.

[13] H. Park, K. Lee, J.-Y. Sohn, C. Suh, and J. Moon, "Hierarchical coding for distributed computing," in *Proc. IEEE Int. Symp. Inf. Theory*, 2018, pp. 1630–1634.

[14] S. Lin and D. J. Costello, *Error Control Coding, Second Edition*. USA: Prentice-Hall, Inc., 2004.

[15] R. Koetter and F. R. Kschischang, "Coding for errors and erasures in random network coding," *IEEE Trans. Inf. Theory*, vol. 54, no. 8, pp. 3579–3591, Aug 2008.

[16] J. A. Cabrera, R. Schmoll, G. T. Nguyen, S. Pandi, and F. H. P. Fitzek, "Softwarization and network coding in the mobile edge cloud for the tactile internet," *Proc. of the IEEE*, vol. 107, no. 2, pp. 350–363, Feb 2019.

[17] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *IEEE Trans. Inf. Theory*, vol. 24, no. 3, pp. 384–386, May 1978.

[18] V. Smith, C.-K. Chiang, M. Sanjabi, and A. Talwalkar, "Federated multi-task learning," in *Proc. 31st Conf. on Neural Inf. Process. Syst.*, Long Beach, CA, Dec. 2017.