A Unified Coded Deep Neural Network Training Strategy based on Generalized PolyDot codes

Sanghamitra Dutta*[1], Ziqian Bai*[2], Haewon Jeong[1], Tze Meng Low[1], Pulkit Grover[1]

***Abstract*—This paper has two main contributions. First, we propose a novel coding technique – Generalized PolyDot – for matrix-vector products that advances on existing techniques for coded matrix operations under storage and communication constraints. Next, we use Generalized PolyDot for the problem of training large Deep Neural Networks (DNNs) using unreliable nodes that are prone to *soft-errors*, e.g., bit flips during computation that produce erroneous outputs. An additional difficulty imposed by the problem of DNN training is that the parameter values (weight matrices) are updated at every iteration, and thus require a prohibitively large encoding cost at every iteration if we naively extend existing coded computing techniques. Thus, we propose a "unified" coded DNN training strategy where we weave coding into the operations of DNN training itself, so that the weight matrices, once initially encoded, remain encoded during updates with negligible encoding/decoding overhead per iteration. Moreover, our strategy can also allow for errors even in the nonlinear step of training. Finally, our coded DNN training strategy is completely decentralized: no assumptions on the presence of a master node are made, which avoids any single point of failure under soft-errors. Our strategy can provide unboundedly better error tolerance than the competing replication strategy and an MDS-code-based strategy [1].**

I. Introduction

DNNs are immensely popular today, with applications such as image processing in safety and time critical computations (e.g. automated cars) and healthcare. Thus reliable training of DNNs is becoming increasingly important. In this paper (expanded version [2]), we propose a unified coded computing technique for error-resilient training of *model parallel*[1] DNNs, based on a new class of codes called Generalized PolyDot.

Our focus is on *soft-errors* that refer to undetected errors, e.g. bit-flips or gate errors in computation, caused by several factors, e.g., exposure of chips to cosmic rays from outer space, manufacturing defects, and storage faults [4]. Ignoring "soft-errors" entirely during the training of DNNs can severely degrade the accuracy of training, as we experimentally observe in [1]. In [1], we also provided some simple coding strategies that leverage Maximum Distance Separable (MDS) codes for model parallel training of DNNs.

Coded computing is a promising solution to the various problems arising from unreliability of processing nodes in parallel and distributed computing, such as straggling [5]. It is a significant step in a long line of work on noisy computing started by von Neumann [6] in 1956, that has been followed upon by Algorithm-Based Fault-Tolerance (ABFT) [7], the predecessor of coded computing (see [2] for expanded survey).

This work proposes a novel coded compputing technique **Generalized PolyDot** for matrix-vector products that generalizes our prior work on PolyDot codes [8], proposed for the problem of multiplying square matrices. In [8], we first demonstrated that the recovery threshold of Polynomial codes [9] can be reduced further using a novel code construction called MatDot. Conceptually, PolyDot codes are a coded matrix-multiplication approach that interpolates between the seminal Polynomial codes [9] (for low communication costs) and MatDot codes [8] (for highest error tolerance). Our proposed **Generalized PolyDot** achieves the same erasure recovery threshold (and hence error tolerance) for matrix-vector products as that obtained in a concurrent work on entangled-polynomial codes [10], proposed for matrix-matrix products.

We utilize Generalized PolyDot to develop a unified coded DNN technique (building on our recent work [1]). However, the problem of DNN training imposes several additional difficulties that we address here:

- **Encoding overhead:** Existing works on coded matrix-vector products (e.g. for computing $\boldsymbol{W}_{N\times N}\boldsymbol{x}_{N\times 1}$) require encoding of the matrix $\boldsymbol{W}$ which is as computationally expensive as the matrix-vector product itself. Thus, these techniques [3], [5], [11] are most useful if $\boldsymbol{W}$ is known in advance and is fixed over a large number of computations so that the encoding cost is amortized. However, when training DNNs, because the parameters *update* at every iteration, a naive extension of existing techniques would require encoding of weight matrices at every iteration and thus introduce an undesirable additional overhead of $\Omega(N^2)$ at every iteration. To address this, we carefully weave coding into operations of DNN training so that an initial encoding of the weight matrices is maintained *across the updates*. Further, to maintain the coded structure, we only need to encode *vectors instead of matrices* at every iteration, thus adding negligible overhead.
- **Master node acting as a single point of failure:** Because of our focus on soft-errors, unlike many coded computing works, we also need to consider a completely decentralized setting, with no master node. This is because a master node can often become a "single point of failure", an important concept in parallel computing. Thus, we allow for even encoding/decoding [12] to be error-prone.
- **Nonlinear activation between layers:** We code the linear operations (matrix-vector products) at each layer separately as they are *the* most critical and complexity-intensive steps in

[1] Carnegie Mellon University, [2] Chinese University of Hong Kong
* Equal Contribution, Contact: sanghamd@andrew.cmu.edu.

[1] *Data parallel* and *model parallel* are two different architectures for training. The former lets each node store and train a replica of the entire DNN on different data and a central parameter server combines their inputs to train a central replica of the DNN. In the latter, different parts of a single DNN are parallelized across nodes. See [3] for coding in data parallel training.

978-1-5386-4780-6/18/$31.00©2018 IEEE

the training of DNNs as compared to other operations such as nonlinear activation or diagonal matrix post-multiplication which are linear in vector length. Moreover, as our implementation is decentralized, every node acts as a replica of the master node, performing encoding/decoding/nonlinear activation/diagonal matrix post-multiplication and helping us detect (and if possible correct) errors in all the steps.

Finally we demonstrate scaling sense advantages of our proposed unified coded DNN strategy over replication and MDS code-based strategies [1] (see Theorem 2).

II. Background and Problem Formulation

We first provide a brief background on DNN training to set the notation. This description is limited; we refer the reader to our full version [2, Appendix A] for a systematic introduction.

Background on DNN training: A DNN with L layers is being trained using backpropagation with Stochastic Gradient Descent with a "batch size" of 1 [2]. The DNN thus consists of L weight matrices, one for each layer (see Fig. 1). At the l-th layer, N_l denotes the number of neurons. Thus, the weight matrix to be trained is of dimension $N_l \times N_{l-1}$. For simplicity of presentation, we assume that $N_l = N$ for all layers.

In every iteration, the DNN (*i.e.* the L weight matrices) is trained based on a single data point and its true label through three stages, namely, feedforward, backpropagation and update, as shown in Fig. 1. At the beginning of every iteration, the first layer accesses the data vector (input for layer 1) from memory and starts the feedforward stage which propagates from layer $l=1$ to L. For a layer, let us denote the weight matrix, input for the layer and backpropagated error for that layer by $\boldsymbol{W}$, $\boldsymbol{x}$ and $\boldsymbol{\delta}$ respectively[2]. The operations performed in layer l during feedforward stage (see Fig. 1a) can be summarized as:

- $[O1]$ Compute matrix-vector product $\boldsymbol{s}=\boldsymbol{W}\boldsymbol{x}$.
- $[C1]$ Compute input for layer $(l+1)$ given by $f(\boldsymbol{s})$ where $f(.)$ is a nonlinear activation function applied elementwise.

At the last layer ($l=L$), the backpropagated error vector is generated by accessing the true label from memory and the estimated label as output of last layer (see Fig. 1b). Then, the backpropagated error propagates from layer L to 1 (see Fig. 1c), also updating the weight matrices at every layer alongside (see Fig. 1d). The operations for the backpropagation stage can be summarized as:

- $[O2]$ Compute matrix-vector product $\boldsymbol{c}^T=\boldsymbol{\delta}^T\boldsymbol{W}$.
- $[C2]$ Compute backpropagated error vector for layer $(l-1)$ given by $\boldsymbol{c}^T\boldsymbol{D}$ where $\boldsymbol{D}$ is a diagonal matrix whose i-th diagonal element depends only on the i-th value of $\boldsymbol{x}$.

Finally, the step in the Update stage is as follows:

- $[O3]$ Update as: $\boldsymbol{W} \leftarrow \boldsymbol{W}+\eta\boldsymbol{\delta}\boldsymbol{x}^T$ where η is the learning rate.

Desirable Parallelization Scheme: We are interested in fully decentralized, model parallel architectures where each layer is parallelized using P nodes for each layer (that can be reused across layers) because the nodes cannot store the entire matrix $\boldsymbol{W}$ for each layer. As the steps $O1$, $O2$ and $O3$ are the most computationally intensive steps at each layer, we restrict ourselves to schemes where these three steps for each layer are parallelized across the P nodes. In such schemes, the steps $C1$ and $C2$ thus become the steps requiring communication as the partial computation outputs of steps $O1$ and $O2$ at one layer are required to compute the input $\boldsymbol{x}$ or backpropagated error $\boldsymbol{\delta}$ for another layer, which is also parallelized across all nodes. We introduce two possible error models here.

Definition 1 (Error Model 1)**.** *Any node can be affected by errors but only during the steps $O1$, $O2$ and $O3$. There are no errors in encoding/decoding/nonlinear activation/diagonal matrix post-multiplication, as they are negligible in computational complexity*[3]*. No assumption is made on the distribution of the errors but the number of errors at each step is bounded.*

Definition 2 (Error Model 2)**.** *Any processing node can be affected by soft-errors at any point during the computation (including encoding/decoding/nonlinear activation/diagonal matrix post-multiplication), and there is no upper bound on the number of errors. For conceptual simplicity, the output of an erroneous node is assumed to be the correct output corrupted by an* ***additive continuous valued random noise****.*

Remark 1. *Error model 1 is a "worst-case" abstraction consistent with finite precision as well as reals (infinite precision). Error model 2 instead allows us to* ***detect*** *the occurrence of errors ("garbage outputs") in a coded computation with probability 1 even if they are too many to be corrected. In practical implementations, our results that hold with probability 1 should be interpreted as holding with high probability (e.g. it is unlikely, but possible, that two erroneous nodes produce the same garbage output). Further, both replication and coding strategies are also able to exploit Error model 2 alike.*

Goal: Our *goal* is to design a unified coded DNN training strategy, denoted by $\mathcal{C}(N,K,P)$, using P nodes such that every node can effectively store only a $\frac{1}{K}$ fraction of the entries of $\boldsymbol{W}$ for every layer. Thus, each node has a total storage constraint of $\frac{LN^2}{K}$ along with negligible additional storage of $o(\frac{LN^2}{K})$ for vectors that are significantly smaller compared to matrices. Additionally it is desirable that all additional communication complexities and encoding/decoding overheads should be negligible in scaling sense compared to the computational complexity of the steps $O1$, $O2$ and $O3$ parallelized across each node, at any layer[4].

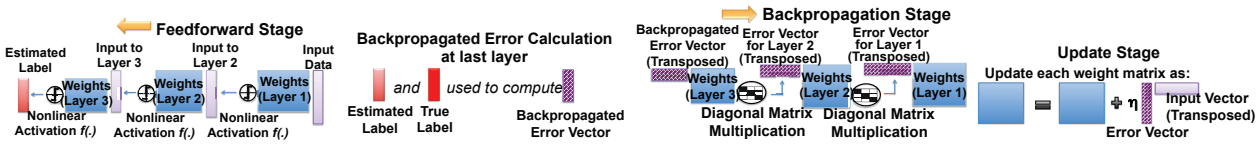
Essentially, we are required to perform coded "post" and "pre" multiplication of the same matrix $\boldsymbol{W}$ with vectors $\boldsymbol{x}$ and $\boldsymbol{\delta}^T$ respectively at each layer, along with all the other operations mentioned in Section II including the update. As outputs are communicated to other nodes at steps $C1$ and $C2$, we would like to be able to correct as many erroneous nodes as possible at these two steps, before moving to another layer.

Definition 3 (Error Tolerances (t_f,t_b))**.** *For any layer l, the error tolerances are (t_f,t_b) if at most t_f and t_b erroneous node outputs can be detected and corrected in steps C_1 and*

[2]Strictly speaking, we should use $\boldsymbol{W}^l$, $\boldsymbol{x}^l$ and $\boldsymbol{\delta}^l$ where l is the index of the layer. However, as the operations are same across layers, we omit the l.

[3]The shorter the computation, the lower is the probability of soft-errors. The occurrence of soft-errors is assumed to be a Poisson process in [13], *i.e.*, the number of errors in an interval has mean proportional to its length.

[4]We are able to compare communication and computational complexities in a scaling sense following [14], even though the constant factor might differ.



(a) Feedforward Stage (b) Transition at last layer (c) Backpropagation Stage (d) Update Stage

Fig. 1. DNN training: (From Left to Right) (a) Feedforward stage - The data vector is passed forward through all the layers (a matrix-vector product followed by a nonlinear activation function $f(.)$ at each layer) producing an estimate of the label vector. (b) Transition - The backpropagated error for the last layer is calculated using the estimated and true label vectors. (c) Backpropagation stage of DNN training - The backpropagated error vector propagates backward across the layers (a matrix-vector product followed by a multiplication with a diagonal matrix) to generate the backpropagated error vector for every layer. (d) Update stage - Alongside, each layer also updates itself using its backpropagated error vector and its own input vector.

C_2 respectively under both Error Models 1 and 2.

Matrix Partitioning Notations: We choose two integers m and n such that $K{=}mn$, and block-partition the matrix $\boldsymbol{W}$ both row-wise and column-wise into $m{\times}n$ blocks, each of size $\frac{N}{m}{\times}\frac{N}{n}$. Let $\boldsymbol{W}_{ij}$ denote the block with row index i and column index j, where $i{=}0,1,...,m{-}1$ and $j{=}0,1,...,n{-}1$. The vectors $\boldsymbol{x}$ and $\boldsymbol{\delta}^T$ are also partitioned into n and m equal parts respectively, denoted by $\boldsymbol{x}_0,\boldsymbol{x}_1,...,\boldsymbol{x}_{n-1}$ and $\boldsymbol{\delta}_0^T,\boldsymbol{\delta}_1^T,...,\boldsymbol{\delta}_{m-1}^T$ respectively. E.g., for $m{=}n{=}2$, the partitioning for $\boldsymbol{\delta}^T$, $\boldsymbol{W}$ and $\boldsymbol{x}$ is:

$$\boldsymbol{\delta}^T{=}\begin{bmatrix}\boldsymbol{\delta}_0^T & \boldsymbol{\delta}_1^T\end{bmatrix},\ \boldsymbol{W}{=}\begin{bmatrix}\boldsymbol{W}_{0,0} & \boldsymbol{W}_{0,1}\\ \boldsymbol{W}_{1,0} & \boldsymbol{W}_{1,1}\end{bmatrix} \text{ and } \boldsymbol{x}{=}\begin{bmatrix}\boldsymbol{x}_0\\ \boldsymbol{x}_1\end{bmatrix}.$$

We would also be partitioning the vectors $\boldsymbol{s}(=\boldsymbol{W}\boldsymbol{x})$ and $\boldsymbol{c}^T(=\boldsymbol{\delta}^T\boldsymbol{W})$ into m and n parts respectively, denoted as $\boldsymbol{s}_0,\boldsymbol{s}_1,...,\boldsymbol{s}_{m-1}$ and $\boldsymbol{c}_0^T,\boldsymbol{c}_1^T,...,\boldsymbol{c}_{n-1}^T$ respectively.

We also let $\boldsymbol{a}(u)$ (or $\boldsymbol{A}(u)$) denote a vector (or matrix) whose every element is a polynomial in scalar variable u, *i.e.*, effectively a polynomial in u whose coefficients are all vectors (or matrices) of the same dimension as $\boldsymbol{a}$ (or $\boldsymbol{A}$).

III. Existing Strategies

Replication ($\mathcal{C}_{\text{rep}}(K,N,P)$) For every layer, the matrix $\boldsymbol{W}$ is block-partitioned across a grid of $m{\times}n$ nodes where $K{=}mn$, and $\frac{P}{mn}$ replicas of this system is created using a total of P nodes (assume mn divides P). For computing $\boldsymbol{s}{=}\boldsymbol{W}\boldsymbol{x}$, the node with grid index (i,j) accesses $\boldsymbol{x}_j$ and computes $\boldsymbol{W}_{ij}\boldsymbol{x}_j$. Then, the first node in every row aggregates and computes the sum $\sum_{j=0}^{n-1}\boldsymbol{W}_{ij}\boldsymbol{x}_j{=}\boldsymbol{s}_i$ for $i{=}0,1,...,m{-}1$. For the example with $m{=}n{=}2$, observe the two sub-vectors of $\boldsymbol{s}$ that are required to be reconstructed:

$$\boldsymbol{s}{=}\begin{bmatrix}\boldsymbol{s}_0\\ \boldsymbol{s}_1\end{bmatrix}{=}\begin{bmatrix}\boldsymbol{W}_{0,0} & \boldsymbol{W}_{0,1}\\ \boldsymbol{W}_{1,0} & \boldsymbol{W}_{1,1}\end{bmatrix}\begin{bmatrix}\boldsymbol{x}_0\\ \boldsymbol{x}_1\end{bmatrix}{=}\begin{bmatrix}\boldsymbol{W}_{0,0}\boldsymbol{x}_0{+}\boldsymbol{W}_{0,1}\boldsymbol{x}_1\\ \boldsymbol{W}_{1,0}\boldsymbol{x}_0{+}\boldsymbol{W}_{1,1}\boldsymbol{x}_1\end{bmatrix}$$

After these computations, all the replicas computing the same sub-vector, *i.e.*, say $\boldsymbol{s}_i$, exchange their computational outputs for error correction. Under Error Model 1, any $t{=}\lfloor\frac{P-mn}{2mn}\rfloor$ errors can be tolerated in the worst case. However under Error Model 2, the probability of two outputs having exactly same error is 0. As long as an output occurs at least twice, it is almost surely the correct output. Thus, any $t{=}\frac{P}{mn}{-}2$ errors can be detected and corrected. Then, the correct sub-vectors ($\boldsymbol{s}_i$'s) are communicated to the respective nodes that require it for generating their input for the next layer, and the sub-matrices stored in the erroneous nodes are **regenerated** by accessing other nodes known to be correct.

Additional Steps: At regular intervals, the system also **checkpoints**, i.e., sends the entire DNN to a *disk* for storage. This disk-storage, although time-intensive to retrieve from, can be assumed to be error-free. Under Error Model 2, if more than t errors occur, then with probability 1 none of the outputs match. The system detects the occurrence of errors even though it is unable to correct them. So, it retrieves the DNN from the disk and reverts the computation to the last checkpoint.

A similar technique is followed for backpropagation. The the node with index (i,j) accesses $\boldsymbol{\delta}_i^T$ and computes $\boldsymbol{\delta}_i^T\boldsymbol{W}_{ij}$. Finally the last node in every column aggregates and computes $\sum_{i=0}^{m-1}\boldsymbol{\delta}_i^T\boldsymbol{W}_{ij}{=}\boldsymbol{c}_j^T$ for $j{=}0,1,...,n{-}1$. Error check occurs similarly. If errors can be corrected, then $\boldsymbol{c}_j^T$'s are communicated to the respective nodes that require it to compute backpropagated error for the next layer, along with $\boldsymbol{x}_j$. Interestingly, after these operations, the node with index (i,j) has $\boldsymbol{x}_j$ and $\boldsymbol{\delta}_i^T$, and is thus able to update itself as $\boldsymbol{W}_{ij}{\leftarrow}\boldsymbol{W}_{ij}{+}\eta\boldsymbol{\delta}_i\boldsymbol{x}_j^T$ respectively.

MDS-code based strategy ($\mathcal{C}_{\text{mds}}(K,N,P)$)**:** Another strategy (details in [1]) is to use two systematic MDS codes to encode the block-partitioned matrix $\boldsymbol{W}$. A $(m{+}2t_f,m)$ systematic MDS code is used to encode these blocks row-wise and a $(n{+}2t_b,n)$ systematic MDS code is used to encode column-wise, so as to correct any t_f and t_b errors in steps $C1$ and $C2$ respectively. The total number of nodes is $P{=}mn{+}2t_fn{+}2t_bn$ for this strategy, of which only mn nodes are used in both steps $O1$ and $O2$. In step $O1$, only $P_f{=}mn{+}2t_fn$ nodes corresponding to the $(m{+}2t_f,m)$ code are active and in step $O2$, only $P_b{=}mn{+}2t_bm$ nodes are active.

IV. Generalized PolyDot codes

Before introducing our coded DNN strategy, we first describe our Generalized PolyDot codes for matrix-vector products. Suppose we are required to perform the matrix-vector product $\boldsymbol{s}{=}\boldsymbol{W}\boldsymbol{x}$ using P nodes, such that every node can only store an $\frac{N}{m}{\times}\frac{N}{n}$ coded or uncoded submatrix ($\frac{1}{K}$ fraction) of $\boldsymbol{W}$. Then, we have the following achievability result.

Theorem 1 (Achievability of Generalized PolyDot)**.** *The Generalized PolyDot codes for computing matrix-vector product $\boldsymbol{W}_{N\times N}\boldsymbol{x}_N$ using P nodes, each storing only an $\frac{N}{m}{\times}\frac{N}{n}$ submatrix, can tolerate atmost $P{-}mn{-}n{+}1$ erasures or $\frac{P-mn-n+1}{2}$ errors under Error Models 1 and 2.*

We let the p-th node ($p{=}0,1,...,P{-}1$) store an encoded block of $\boldsymbol{W}$ which is a polynomial in u and v

$$\tilde{\boldsymbol{W}}(u,v){=}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\boldsymbol{W}_{ij}u^iv^j \tag{1}$$

evaluated at $(u,v)=(a_p,b_p)$. Each node also block-partitions $\boldsymbol{x}$ into n equal parts, and encodes them using the polynomial

$$\tilde{\boldsymbol{x}}(v)=\sum_{l=0}^{n-1}\boldsymbol{x}_l v^{n-l-1}. \quad (2)$$

evaluated at $v=b_p$. Then, each node *performs* the matrix-vector product $\tilde{\boldsymbol{W}}(a_p,b_p)\tilde{\boldsymbol{x}}(b_p)$ which effectively results in the evaluation, at $(u,v)=(a_p,b_p)$, of the following polynomial:

$$\tilde{\boldsymbol{s}}(u,v)=\tilde{\boldsymbol{W}}(u,v)\tilde{\boldsymbol{x}}(v)=\sum_{l=0}^{n-1}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\boldsymbol{W}_{ij}\boldsymbol{x}_l u^i v^{n-l+j-1} \quad (3)$$

even though the node is *not explicitly evaluating* it from all its coefficients. Now, fixing $l=j$, observe that the coefficient of $u^i v^{n-1}$ for $i=0,1,...,m-1$ turns out to be $\sum_{j=0}^{n-1}\boldsymbol{W}_{ij}\boldsymbol{x}_j=\boldsymbol{s}_i$. Thus, these m coefficients constitute the m sub-vectors of $\boldsymbol{s}=\boldsymbol{W}\boldsymbol{x}$. Therefore, $\boldsymbol{s}$ can be recovered at any node if it can reconstruct these m coefficients of the polynomial $\tilde{\boldsymbol{s}}(u,v)$ in (3). Let us illustrate this for the case where $m=n=2$. Consider the following polynomial:

$$\begin{aligned}\tilde{\boldsymbol{s}}(u,v)&=(\boldsymbol{W}_{0,0}+\boldsymbol{W}_{1,0}u+\boldsymbol{W}_{0,1}v+\boldsymbol{W}_{1,1}uv)(\boldsymbol{x}_0v+\boldsymbol{x}_1)\\&=\boldsymbol{W}_{0,0}\boldsymbol{x}_1+\boldsymbol{W}_{1,0}\boldsymbol{x}_1u+\boldsymbol{W}_{0,1}\boldsymbol{x}_0v^2+\boldsymbol{W}_{1,1}\boldsymbol{x}_0uv^2\\&+\underbrace{(\boldsymbol{W}_{0,0}\boldsymbol{x}_0+\boldsymbol{W}_{0,1}\boldsymbol{x}_1)}_{\boldsymbol{s}_0}v+\underbrace{(\boldsymbol{W}_{1,0}\boldsymbol{x}_0+\boldsymbol{W}_{1,1}\boldsymbol{x}_1)}_{\boldsymbol{s}_1}uv \quad (4)\end{aligned}$$

We use the substitution $u=v^n$ to convert $\tilde{\boldsymbol{s}}(u,v)$ into a polynomial in a single variable. Some of the unwanted coefficients align with each other (e.g. u and v^2 in (4)), but the coefficients of $u^iv^{n-1}=v^{ni+n-1}$ stay the same, *i.e.*, $\boldsymbol{s}_i$ for $i=0,1,...,m-1$. The resulting polynomial is of degree $mn+n-2$. Thus, all the coefficients of this polynomial can be reconstructed from P distinct evaluations of this polynomial at P nodes, if there are atmost $P-mn-n+1$ erasures or $\frac{P-mn-n+1}{2}$ errors [2].

V. Unified Coded DNN Training Strategy

Here, we propose an initial encoding scheme for $\boldsymbol{W}$ at each layer such that the same encoding allows us to perform coded"post" and "pre" multiplication of $\boldsymbol{W}$ with vectors $\boldsymbol{x}$ and $\boldsymbol{\delta}^T$ respectively at each layer in every iteration. The key idea is that we encode $\boldsymbol{W}$ only for the first iteration. For all subsequent iterations, we encode and decode vectors (hence complexity $o(\frac{N^2}{K})$ as we show in Theorem 3) instead of matrices. As we will show, the encoded weight matrix $\boldsymbol{W}$ is able to update itself, maintaining its coded structure.

Initial Encoding of $\boldsymbol{W}$: Every node receives an $\frac{N}{m}\times\frac{N}{n}$ sub-matrix (or block) of $\boldsymbol{W}$ encoded using Generalized PolyDot. For $p=0,1,...,P-1$, node p stores $\tilde{\boldsymbol{W}}_p:=\tilde{\boldsymbol{W}}(u,v)|_{u=a_p,v=b_p}$ (recall (1)) at the beginning of the training which has $\frac{N^2}{K}$ entries. *Encoding of matrix is done only in the first iteration.*

Feedforward Stage: Assume that the entire input $\boldsymbol{x}$ to the layer is available at every node at the beginning of step $O1$ (this assumption is justified at the end of this section). Also assume that the updated $\tilde{\boldsymbol{W}}_p$ of the previous iteration is available at every node (this assumption will be justified when we show that the encoded sub-matrices of $\boldsymbol{W}$ are able to update themselves, preserving their coded structure).

For $p=0,1,...,P-1$, node p block partitions $\boldsymbol{x}$ and generates the codeword $\tilde{\boldsymbol{x}}_p:=\tilde{\boldsymbol{x}}(v)|_{v=b_p}$ (see (2)). Next, each node *performs* the matrix-vector product: $\tilde{\boldsymbol{s}}_p=\tilde{\boldsymbol{W}}_p\tilde{\boldsymbol{x}}_p$ and sends this product (polynomial evaluation) to every other node[5] where some of these products may be erroneous. Now, if every node can still decode the coefficients of u^iv^{n-1} for $i=0,1,...,m-1$), then it can successfully decode s.

We actually use one of the substitutions $u=v^n$ or $v=u^m$ (elaborated in Section VI and [2, Appendix B]), to convert $\tilde{\boldsymbol{s}}(u,v)$ into a polynomial in a single variable and then use standard decoding techniques [2, Appendix B] to interpolate the coefficients of a polynomial in one variable from its evaluations at P arbitrary points when some evaluations have an additive error. Once $\boldsymbol{s}$ is decoded, the nonlinear function $f(.)$ is applied element-wise to generate the input for the next layer. This also makes $\boldsymbol{x}$ available at every node at the start of the next feedforward layer, justifying our assumption.

Regeneration: Under both Error Models 1 and 2, each node can not only correct t_f erroneous nodes but also locate which nodes were erroneous [2, Appendix B]. Thus, the encoded $\boldsymbol{W}$ stored at those nodes are **regenerated**[6] by accessing some of the nodes that are known to be correct.

Additional Steps: Similar to replication and MDS code based strategy, the DNN is **checkpointed** at a disk at regular intervals. If there are more errors than the error tolerance, the nodes are unable to decode correctly. However under Error Model 2, as the error is assumed to be additive and drawn from real-valued, continuous distributions, the occurrence of errors is still detectable [2] even though they cannot be located or corrected, and thus the entire DNN can again be *restored from the last checkpoint*.

To allow for **decoding errors** under Error Model 2, we need to include one more verification step where all nodes exchange their assessment of node outputs, *i.e.*, a list of nodes that they found erroneous and compare (additional overhead of $\Theta(P^2(\log P))$ [2, Appendix C]). If there is a disagreement at one or more nodes during this process, we assume that there has been errors during the decoding, and the entire neural network is restored from the last checkpoint. Because the complexity of this verification step is low in scaling sense compared to encoding/decoding or communication (because it does not depend on N), we assume that it is error-free since the probability of soft-errors occurring within such a small duration is negligible as compared to other computations of longer durations.

Backpropagation Stage: The backpropagation stage is very similar to the feedforward stage. The backpropagated error $\boldsymbol{\delta}^T$ is available at every node. Each node partitions the row-vector $\boldsymbol{\delta}^T$ into m equal parts and encodes them using the polynomial:

$$\tilde{\boldsymbol{\delta}}^T(u)=\sum_{l=0}^{m-1}\boldsymbol{\delta}_l^T u^{m-l-1}. \quad (5)$$

[5]Pessimistically, we assume that every node first multi-casts its own output to all P nodes in $\Theta(\log P)$ rounds, and then this is repeated for P nodes allowing the communication overhead to be as high as $\Theta(\frac{N}{m}P\log P)$. This complexity might be reduced using other all-to-all communication protocols.

[6]The encoded matrix at any node is the evaluation of a polynomial whose coefficients correspond to the original sub-matrices $\boldsymbol{W}_{ij}$. Thus, the number of nodes required by an error-prone node is the degree of this polynomial $+1$. Substituting $u=v^n$ (alternatively, $v=u^m$), this degree is $mn-1$, and thus an error-prone node needs to access mn correct nodes to regenerate itself.

For p=0,1,...,$P-1$, the p-th node *evaluates* $\tilde{\boldsymbol{\delta}}^T(u)$ at $u=a_p$, yielding $\tilde{\boldsymbol{\delta}}_p^T=\tilde{\boldsymbol{\delta}}^T(a_p)$. Next, it *performs* the computation $\tilde{\boldsymbol{c}}_p^T=\tilde{\boldsymbol{\delta}}_p^T\tilde{\boldsymbol{W}}_p$ and sends the product to all the other nodes, of which some products may be erroneous. Consider the polynomial:

$$\tilde{\boldsymbol{c}}^T(u,v)=\tilde{\boldsymbol{\delta}}^T(u)\tilde{\boldsymbol{W}}(u,v)=\sum_{l=0}^{m-1}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\boldsymbol{\delta}_l^T\boldsymbol{W}_{ij}u^{m-l+i-1}v^j$$

The products computed at each node effectively result in the evaluations of this polynomial $\tilde{\boldsymbol{c}}^T(u,v)$ at $(u,v)=(a_p,b_p)$. Similar to feedforward stage, each node is required to decode the coefficients of $u^{m-1}v^j$ in this polynomial for j=0,1,...,$n-1$ to reconstruct $\boldsymbol{c}^T$. The vector $\boldsymbol{c}^T$ is used to compute the backpropagated error for the consecutive, *i.e.*, $(l-1)$-th layer.

Update Stage: The key part is *updating* the coded $\boldsymbol{W}_p$. Observe that since $\boldsymbol{x}$ and $\boldsymbol{\delta}$ are both available at each node, it can encode the vectors as $\sum_{i=0}^{m-1}\boldsymbol{\delta}_i u^i$ and $\sum_{j=0}^{n-1}\boldsymbol{x}_j^T v^j$ at $u=a_p$ and $v=b_p$ respectively, and then update itself as follows:

$$\tilde{\boldsymbol{W}}_p\leftarrow\tilde{\boldsymbol{W}}_p+\eta(\sum_{i=0}^{m-1}\boldsymbol{\delta}_i a_p^i)(\sum_{j=0}^{n-1}\boldsymbol{x}_j^T b_p^j)$$

$$=\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\underbrace{(\boldsymbol{W}_{ij}+\eta\boldsymbol{\delta}_i\boldsymbol{x}_j^T)}_{\text{Update of }\boldsymbol{W}_{ij}}a_p^i b_p^j \tag{6}$$

The update step *preserves* the coded nature of the weight matrix, with negligible additional overhead (see Theorem 3). Errors occurring in the update stage corrupt the updated submatrix without being immediately detected as there is no output produced. The errors exhibit themselves only after step $O1$ in the next iteration at that layer, when that particular submatrix is used to produce an output again. Thus, they are detected (and if possible corrected) at $C1$ of next iteration.

VI. Comparison with existing strategies

We compare the worst case error tolerance of $\mathcal{C}_{\text{GP}}(K,N,P)$ with $\mathcal{C}_{\text{mds}}(K,N,P)$ and $\mathcal{C}_{\text{rep}}(K,N,P)$ in Theorem 2 below.

Theorem 2 (Error tolerances (t_f,t_b))**.** *The error tolerances (t_f,t_b) at each layer for the three strategies $\mathcal{C}_{\text{GP}}(K,N,P)$, $\mathcal{C}_{\text{mds}}(K,N,P)$ and $\mathcal{C}_{\text{rep}}(K,N,P)$ are given by Table I.*

TABLE I
Error Tolerances (t_f,t_b) under fixed number of nodes P

Strategy	Error Tolerance in Step $C1$ (t_f)	Error Tolerance in in Step $C2$ (t_b)
$\mathcal{C}_{\text{GP}}(K,N,P)$ with $u=v^n$	$\frac{P-mn-n+1}{2}$	$\frac{P-2mn+n}{2}$
$\mathcal{C}_{\text{GP}}(K,N,P)$ with $v=u^m$	$\frac{P-2mn+m}{2}$	$\frac{P-mn-m+1}{2}$
$\mathcal{C}_{\text{mds}}(K,N,P)$ where $P=P_f+P_b-mn$	$\frac{P_f-mn}{2n}\left(\leq\frac{P-mn}{2n}\right)$	$\frac{P_b-mn}{2m}\left(\leq\frac{P-mn}{2m}\right)$
$\mathcal{C}_{\text{rep}}(K,N,P)$	$\frac{P}{mn}-2$	$\frac{P}{mn}-2$

Remark 2. *Strictly speaking, we need a floor function $\lfloor.\rfloor$ applied to all of the expressions and $mn|P$ for replication.*

Remark 3. *One might prefer $t_f>t_b$ because at step $C1$ all errors from both steps $O1$ of the current iteration and $O3$ of the last iteration are corrected along with low-complexity intermediate steps. However, at step $C2$, only errors at $O2$ are corrected along with low-complexity intermediate steps.*

Corollary 1 (Scaling Sense Comparison)**.** *Consider the regime $m=n=\sqrt{K}$. Then the ratio of t_f (or t_b) for $\mathcal{C}_{\text{GP}}(K,N,P)$ with $\mathcal{C}_{\text{mds}}(K,N,P)$ and $\mathcal{C}_{\text{rep}}(K,N,P)$ scales as $\Theta(\sqrt{K})$ and $\Theta(K)$ respectively as $P\to\infty$.*

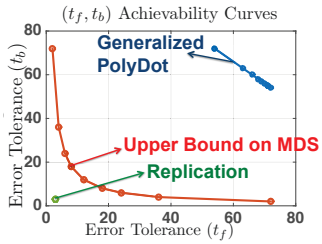


Fig. 2. Error tolerance region: We choose P=180, K=36 and vary m and n. For MDS-code based strategy, we plot an upper bounds on t_f,t_b using $P_f,P_b\leq P$. Generalized PolyDot (with $u=v^n$) achieves the best (t_f,t_b) trade-off. Choosing $v=u^m$ also gives same curve only interchanging (t_f,t_b).

All proofs are in [2, Appendix B]. In Fig. 2, we show that Generalized PolyDot achieves the best (t_f,t_b) trade-off compared to the other existing schemes. Now we formally show that $\mathcal{C}_{\text{GP}}(K,N,P)$ satisfies the desired properties of adding negligible overhead at each node in Theorem 3.

Theorem 3. *For a $\mathcal{C}_{\text{GP}}(K,N,P)$ in feedforward (or backpropagation) stage at any layer, the ratio of the total complexity of encoding/decoding and communication to the matrix-vector product tends to 0 as $K,N,P\to\infty$ if the number of nodes satisfy $P^4=o(N)$.*

The proof is provided in [2, Appendix C]. For this proof, we assume a pessimistic bound ($\Theta(P^3)$) on the decoding of a code of block length P under errors, based on sparse reconstruction algorithms [15]. Reduction of decoding complexity using other algorithms would also relax the condition of Theorem 3.

References

[1] S. Dutta, Z. Bai, T. M. Low, and P. Grover, "Codenet: Training Large Neural Networks in presence of Soft-Errors," *Submitted*, 2018.

[2] "Full version." [Online]. Available: sites.google.com/site/sanghamitraweb/academic-articles

[3] R. Tandon *et al.*, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *International Conference on Machine Learning*, 2017.

[4] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[5] K. Lee *et al.*, "Speeding Up Distributed Machine Learning Using Codes," *IEEE Trans. on Inf. Theory*, vol. PP, no. 99, pp. 1–1, 2017.

[6] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, vol. 34, pp. 43–98, 1956.

[7] K. H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[8] M. Fahim *et al.*, "On the Optimal Recovery Threshold of Coded Matrix Multiplication," in *Comm., Control, and Computing (Allerton)*, 2017.

[9] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," in *Advances In Neural Information Processing Systems (NIPS)*, 2017.

[10] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," *arXiv preprint arXiv:1801.07487*, 2018.

[11] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *Advances In Neural Information Processing Systems (NIPS)*, 2016.

[12] M. G. Taylor, "Reliable Information Storage in Memories Designed from Unreliable Components," *Bell Syst. Tech. J.*, vol. 47, no. 10, pp. 2299–2337, 1968.

[13] X. Li *et al.*, "A memory soft error measurement on production systems." in *USENIX Annual Technical Conference*, 2007.

[14] R. A. van de Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," Austin, TX, USA, Tech. Rep., 1995.