A Learning-Based Approach to Approximate Coded Computation

Navneet Agrawal[1*], Yuqin Qiu[2*], Matthias Frey[1], Igor Bjelakovic[3], Setareh Maghsudi[3,4],
Slawomir Stanczak[1,3], and Jingge Zhu[2]

[1] Technische Universität Berlin, Germany
[2] Department of Electrical and Electronic Engineering, University of Melbourne, Victoria, Australia
[3] Fraunhofer Heinrich Hertz Institute, Berlin, Germany
[4] Department of Computer Science, University of Tübingen, Germany

Abstract—**Lagrange coded computation (LCC) is essential to solving problems about matrix polynomials in a coded distributed fashion; nevertheless, it can only solve the problems that are representable as matrix polynomials. In this paper, we propose AICC, an AI-aided learning approach that is inspired by LCC but also uses deep neural networks (DNNs). It is appropriate for coded computation of more general functions. Numerical simulations demonstrate the suitability of the proposed approach for the coded computation of different matrix functions that are often utilized in digital signal processing.**

I. Introduction

Coded Computation (CC) refers to a class of distributed computation schemes in which an encoder injects redundancy into the input data before it is distributed to the worker machines to perform the desired computation task. This allows reconstruction of the full computation result with a decoding step even if not all the workers have returned computation results. Therefore, CC schemes can reduce latency and increase reliability in scenarios in which the worker nodes are, e.g., subject to random failures or straggling [1].

The CC schemes with the recovery of the exact computation results are, for instance, applicable to matrix-matrix multiplication [2], [3] and the computation of matrix polynomials [4], [5]. Some papers have also proposed CC schemes for these problems that can work with a significantly lower number of worker results in the decoding step. This comes at the cost of recovering the computation result only approximately with a certain prescribed accuracy [6]–[8]. Another line of research considers schemes that have a soft recovery threshold in the sense that they do not need a minimum number of worker results to obtain an estimate. Instead, they obtain estimates with increasing accuracy as more and more of these results become available [9]–[11]. Approximate coded computation

is relevant for many applications, including those pertaining to approximate machine learning (ML) tasks using deep neural network (DNN) models.

Many existing papers propose schemes specifically for the training stage of DNNs that deal with erasures as well as with maliciously injected incorrect computation results (seminal works include [12], [13]). References [14]–[16] apply CC to the inference stage of ML computation tasks, and follow the traditional CC paradigm that divides the scheme into the three stages of encoding, computing, and decoding. These papers propose methods for "learning" a suitable erasure code, wherein, one approach jointly trains an auto-encoder for encoding and decoding operations, whereas, in another, the computation corresponding to the encoded data is a specially trained neural network. In both approaches, not only the system is constrained to a single erasure, but the performance also degrades significantly with increasing number of inputs.

In this work, we aim to expand the class of functions compatible with the approximate CC schemes to include non-polynomial functions other than, but not excluding, the evaluations of the inference stage of neural networks. For numerical evaluations, we focus on the approximate CC computations of: (P_1) the eigenvalues, (P_2) the dominant eigenvector corresponding to the largest absolute eigenvalue, (P_3) the matrix exponential, and (P_4) the matrix determinant of a square matrix. These problems are of particular interest in the context of wireless communications [17], [18].

To the best of our knowledge, the CC schemes for these functions have not yet been proposed in the state-of-the-art research. Our approach is substantially inspired by the Lagrange Coded Computation (LCC) [4], but it extends the idea by using neural networks to achieve the following improvements:

- The class of computable functions are no longer limited to the matrix polynomials.

- The proposed scheme guarantees a recovery threshold, which is a design parameter. In particular, its choice does not depend on the number of computation inputs directly.

- Approximate function computation allows for a judicious trade-off between the accuracy and the computation time.

II. Notation and Preliminaries

We consider a system consisting of a *master* and N *workers*. Given a function $f : \mathcal{D} \to \mathcal{S}$, where $\mathcal{D} \subset \mathbb{R}^{M \times M}$ and $\mathcal{S}$ is some finite dimensional Euclidean space, the master is required to compute function values at points gathered in the set $\mathcal{X} \triangleq \{\mathbf{X}_1, \ldots, \mathbf{X}_K\}$, where $\mathbf{X}_k \in \mathcal{D}$ for all $k = 1, \ldots, K$. For this purpose, the master can distribute the computations to the workers.

In coded computation, the master node takes the input data $\mathcal{X}$ to generate N encoded data $\tilde{\mathbf{X}}_i, i = 1, \ldots, N$. The worker i performs computation on the encoded data $\tilde{\mathbf{X}}_i$ and returns the result to the master. The coded computation scheme allows the master to recover all desired function values $f(\mathbf{X}_k)$, $k = 1, \ldots, K$ upon receiving the computation results from any subset of $R \leq N$ workers. The smallest value of R for which this is possible is called the *recovery threshold* of the scheme. In this way, the CC schemes can mitigate the impact of straggling or failing worker.

The LCC [4] scheme is a coded computation scheme for computing polynomial functions f, say, of degree d. The functioning principle of the LCC can be summarized as follows (see [4] for details): The data is encoded to N matries $\tilde{\mathbf{X}}_i$, $i = 1, \ldots, N$, each of which amounts to the evaluation of a Lagrange polynomial of degree $K - 1$ at distinct scalars α_n. The worker $n \in \{1, \ldots, N\}$ then computes the desired polynomial f on the encoded data $\tilde{\mathbf{X}}_n$. The computation result $f(\tilde{\mathbf{X}}_n)$ can be viewed as the evaluation of a composite polynomial of degree $(K - 1)d$, formed by the Lagrange polynomial and the polynomial function f at a point α_n. Hence, to recover this polynomial, the master only requires $R = (K - 1)d + 1$ results from the workers. The desired function values can then be obtained by evaluating the polynomial at suitable points.

Although LCC is optimal in terms of the recovery threshold, it suffers from two crucial limitations: (i) the function f is restricted to be a polynomial of the input in LCC; and (ii) the recovery threshold R of LCC grows proportionally with the number of inputs K and degree d of the polynomial f.

The core motivation for this work is to explore the possibility of extending the idea of LCC to general functions, while ensuring a recovery threshold. Roughly speaking, we replace both the encoding and the computation operations in LCC by deep neural networks. Particularly, these operations are carefully designed such that the computation results from workers are evaluations of a certain polynomial (reminiscent of LCC or any polynomial-based CC scheme), hence only polynomial interpolation is needed for the recovery. In the sequel, we will discuss how our construction can overcome the limitations of LCC mentioned above to some degree. However, we point out that we only aim to recover the result *approximately*, whereas LCC and many other coded computation schemes guarantees a perfect recovery (up to numerical stability issues in the recovery phase).

III. AI-aided Coded Computation Scheme

We propose an *AI-aided coded computation* (AICC) scheme that utilizes a learning-based approach for coded computa-
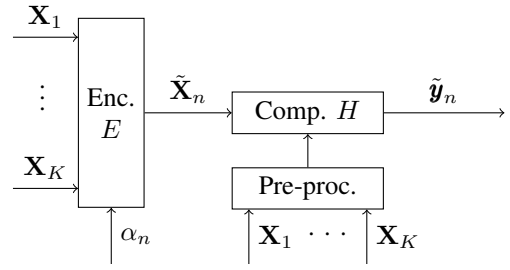

Fig. 1: Encoding and computation operation for worker n.

tion, applicable to a large class of functions including non-polynomials. AICC employs a system design that inherits the recovery-through-interpolation aspect of the LCC, but ensures a fixed recovery threshold that does not directly depend on the number of inputs or polynomial degree of the function. Note that although AICC can only provide an approximate solution in general, in Section IV, we show that our design achieves a sufficient level of accuracy. Since an approximate solution is good enough in many practical applications, AICC is relevant in a wide range of applications.

In the following, we first describe the system architecture and then the training procedure of the proposed scheme.

A. System architecture and operation

The system includes three operations: *encoding*, *computation*, and *decoding*. The master performs the encoding and the decoding operations while the workers do the computations. In the following, we describe each of these operations in detail. Figure 1 depicts the general structure of the encoding and computation operations for one worker.

Encoding operation: Figure 2 shows the structure of the encoding function. Given an integer $G > 0$, the encoder $E : (\mathcal{D})^K \times \mathbb{R} \to \mathbb{R}^{M \times M}$ is given by

$$E(\mathcal{X}, \alpha) \triangleq \sum_{g=0}^{G} \mathbf{U}_g \, \alpha^g. \tag{1}$$

The coefficients $\mathbf{U}_g \in \mathbb{R}^{M \times M}$, $g = 0, \ldots, G$, are obtained via the functions $\Gamma_g : (\mathcal{D})^K \to \mathbb{R}^{M \times M}$ such that $\mathbf{U}_g \triangleq \Gamma_g(\mathcal{X})$. The functions Γ_g, $g = 0, \ldots, G$ are DNNs whose training procedure will be discussed in Section III-B. Given the input set $\mathcal{X}$, the function $e(\cdot) \triangleq E(\mathcal{X}, \cdot)$ is a degree G polynomial. Integer G is a design parameter that, in contrast to the LCC, does not directly depend on the number of input matrices K. Encoded matrices are obtained for N distinct scalars $\{\alpha_1, \ldots, \alpha_N\}$ as:

$$(\forall n \in \{1, \ldots, N\}) \qquad \tilde{\mathbf{X}}_n \triangleq e(\alpha_n) \in \mathbb{R}^{M \times M}, \tag{2}$$

where $\alpha_n \in \mathbb{R}$, for all $n = 1, \ldots, N$. Each encoded matrix is then sent to one of the N workers for computation.

Computation operation H: Figure 3 shows the structure of the computation function. Worker n performs the computation

operation on the encoded matrix $\tilde{\mathbf{X}}_n$. Given an integer $P > 0$, the computation operation $H : (\mathcal{D})^K \times \mathbb{R}^{M \times M} \to \mathbb{R}^V$ yields

$$H(\mathcal{X}, \tilde{\mathbf{X}}) \triangleq \sum_{p=0}^{P} \mathbf{V}_p \, \text{Vec}(\tilde{\mathbf{X}}^p), \tag{3}$$

where $\text{Vec}(\mathbf{X})$ vectorizes matrix $\mathbf{X}$ by stacking its columns. The integer V represents the number of coordinates in the range space $\mathcal{S}$ of the desired function f, and the integer P is a design parameter that does not directly depend on f. The coefficients $\mathbf{V}_p \in \mathbb{R}^{V \times M^2}$, $p = 0, \dots, P$, are obtained via DNNs $\Lambda_p : (\mathcal{D})^K \to \mathbb{R}^{V \times M^2}$ as $\mathbf{V}_p \triangleq \Lambda_p(\mathcal{X})$. Notice that for a given dataset $\mathcal{X}$, the operation H is common to all the workers. Hence, the coefficients $\{\mathbf{V}_p\}$ can be computed by the master, or by any one of the workers, and broadcasted to all workers. Given coefficients $\mathbf{V}_p$, we use $h(\tilde{\mathbf{X}}_n) \triangleq H(\mathcal{X}, \tilde{\mathbf{X}}_n)$ to denote the function that represents the computation operation. It takes encoded data $\tilde{\mathbf{X}}_n$ as input and outputs the computation result $\tilde{\boldsymbol{y}}_n \triangleq h(\tilde{\mathbf{X}}_n)$.

Decoding process: Given a scalar $\alpha \in \mathbb{R}$ and a corresponding encoded data vector $\tilde{\mathbf{X}}$, worker output $\tilde{\boldsymbol{y}} = h(\tilde{\mathbf{X}}) = h(e(\alpha))$ can be viewed as a polynomial evaluated at the point α, since

$$h(e(\alpha)) = \sum_{p=0}^{P} \mathbf{V}_p \, \text{Vec}\left[\left(\sum_{g=0}^{G} \mathbf{U}_g \, \alpha^g \right)^p \right] = \sum_{l=0}^{GP} \boldsymbol{\gamma}_l \, \alpha^l, \tag{4}$$

where the last equality follows by taking scalars of the form α^{gp} out of the $\text{Vec}(\cdot)$ operation. Hence, the function $h \circ e$ is a polynomial of degree $GP \equiv \deg(e)\deg(h)$. In (4), the coefficients $\boldsymbol{\gamma}_l$, $l = 0, \dots, GP$, belong to $\mathbb{R}^V$. Therefore, the number of data points required to recover $v \triangleq h \circ e$ is $R = GP + 1$. The recovery can be performed, for example, using Lagrange interpolation. This integer R is the recovery threshold of the proposed AICC scheme.

To retrieve the final results, the master evaluates v at scalars $\beta_1, \dots, \beta_K$, which are distinct real numbers that are chosen in the training phase, and fixed for the given problem. This aspect is different from LCC where these scalars can be chosen freely by the master at the time of encoding. The final result of the decoding process is given by

$$(\forall k \in \{1, \dots, K\}) \qquad \hat{\boldsymbol{f}}_k \triangleq h(e(\beta_k)) \in \mathbb{R}^V. \tag{5}$$

The master obtains the results $\{\hat{\boldsymbol{f}}_1, \dots, \hat{\boldsymbol{f}}_K\}$ as approximations for $\{f(\mathbf{X}_1), \dots, f(\mathbf{X}_K)\}$. The accuracy of this approximation, measured based on a cost function, is assured by the training procedure described in the next section.

B. Training

The encoding and decoding operations in the AICC scheme described in Section III-A consist of DNNs $\{\Gamma_0, \dots, \Gamma_G, \Lambda_0, \dots, \Lambda_P\}$, each with learnable parameters. These parameters are trained using a training dataset such that the cost of approximating $f(\mathbf{X}_k)$ with $\hat{\boldsymbol{f}}_k$ is minimized for all $k \in \{1, \dots, K\}$ and all input datasets. Distinct real-valued scalars β_k, $k = 1, \dots, K$, are chosen at the beginning. They remain fixed during the entire training and operation phase.
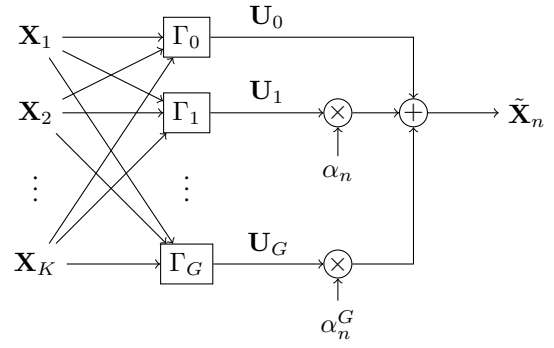

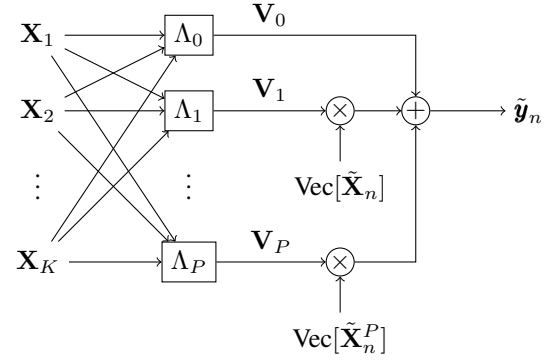Fig. 2: Encoding Function at the master.

Fig. 3: Computation Function of worker n.

The choice of cost function depends on the application, but it should be continuous and differentiable with respect to the learnable parameters appearing in the DNN functions. However, one can select the specific architecture of the DNNs and the training procedure freely. We discuss these details in Section IV for various functions.

Figures 2 and 3 respectively show the input-output structure of the encoding and computation operations with corresponding DNNs. The training consists of forward and backward passes over the system to update the parameters using the stochastic gradient descent (SGD) algorithm. In each forward pass, given a dataset $\mathcal{X}$ and a scalar β_k, the sequence of encoding, computation, and decoding operations result in the output $\hat{\boldsymbol{f}}_k = h(e(\beta_k))$. Then, in the backward pass, the parameters are updated based on the gradients of the cost w.r.t. the parameters, evaluated using the output $\hat{\boldsymbol{f}}_k$ and the target value $\boldsymbol{f}_k \triangleq \text{Vec}(f(\mathbf{X}_k))$.

One instance of the training data consists of inputs $\{\mathbf{X}_1, \dots, \mathbf{X}_K\}$, distinct scalars $\{\beta_1, \dots, \beta_K\}$, and corresponding target labels $\{\boldsymbol{f}_1, \dots, \boldsymbol{f}_K\}$. The inputs $\{\mathbf{X}_1, \dots, \mathbf{X}_K\}$ corresponding to every training instance are sampled i.i.d. from a given distribution which is representative of the true distribution of inputs to the function f.

IV. SIMULATIONS

In this section, we apply the proposed AICC scheme for coded computation of four frequently-used matrix functions in

wireless communication and digital signal processing applications. In the following, we present details of those functions and elaborate on the data generation process. We also discuss relevant implementation issues highlighting unique aspects related to the design and training of DNNs. Numerical results appear at the end, accompanied by a discussion on the effect of various design parameters on the performance.

A. Applied problems

We consider the problems of computation of (P_1) the *eigenvalues*, (P_2) the *eigenvector corresponding to the largest absolute eigenvalue*, (P_3) the *matrix exponential*, and (P_4) the *determinant* of real-valued square matrices. The motivation for our work stems from applications in wireless communications. For example, optimal power allocation [17] and beamforming [18] in modern communication systems require estimating the largest absolute eigenvalue and the dominant eigenvector of large number of matrices in real-time.

In the training phase, the scheme learns to solve a regression problem for function approximation using a supervised learning approach. In this framework, functions with an unbounded range are generally difficult to learn. Thus, for the present simulations, we design the input distribution for each problem to obtain a bounded range for the functions. Nevertheless, in principle, the proposed scheme applies to any function with an arbitrary domain and range for which a system with the structure described in Section III is learnable to approximate the function with the desired accuracy.

For problem P_1, we generate an input sample as $\mathbf{X} \triangleq \frac{1}{2}(\mathbf{A} + \mathbf{A}^T)$, where $\mathbf{A}$ is drawn uniformly from $[-1,1]^{M \times M}$. For problem P_2, we similarly generate an input sample as $\mathbf{X} \triangleq \frac{1}{2}(\mathbf{A} + \mathbf{A}^T)$; however, here, $\mathbf{A}$ is drawn uniformly from $[0,1]^{M \times M}$. We make the eigenvector in P_2 unique by normalizing the output to a vector of unit norm and ensuring that the first nonzero component is positive. For problem P_3, the input sample is generated as $\mathbf{X} \triangleq \mathbf{A}/\|\mathbf{A}\|_{\mathrm{op}}$ with $\mathbf{A}$ drawn uniformly from $[0,1]^{M \times M}$. In addition to keeping the exponential bounded, this ensures the numerical stability of the algorithm used to compute the ground truth in training [19]. Here, $\|\cdot\|_{\mathrm{op}}$ denotes the operator norm of matrices.

The matrix determinant in P_4 is bounded by generating diagonally-dominant matrices. More precisely, we generate the input matrices as $\mathbf{X} \triangleq \mathbf{I} - \mathbf{E}$, where $\mathbf{I}$ is the identity matrix, and $\mathbf{E}$ has zeros on the diagonal and non-diagonal elements sampled uniformly from $[-2.0/M, 2.0/M]$ [20], [21]. The dimensionality of the outputs in problems P_i, for $i = 1, \ldots, 4$, are, respectively, M, M, M^2 and 1. For simulations, we compute the true values of the functions by using the Python's Numpy and Scipy libraries. For problems P_1, P_3, and P_4, we use the cost function $C(\hat{\boldsymbol{f}}, \boldsymbol{f}) := \|\hat{\boldsymbol{f}} - \boldsymbol{f}\|$ for training, where $\boldsymbol{f} \triangleq \mathrm{Vec}(f(\mathbf{X}))$. We use $\|\cdot\|$ to denote the Frobenius norm for matrices and vector norm for vectors. For the eigenvector problem P_2, we add an additional term in the cost function to influence the network in learning a vector of unit norm, i.e., $\tilde{C}(\hat{\boldsymbol{f}}, \boldsymbol{f}) := C(\hat{\boldsymbol{f}}, \boldsymbol{f}) + 5 \left(\|\hat{\boldsymbol{f}}\| - 1.0\right)^2$.

B. Implementation details

Below we explicitly describe the DNN architecture and system design. For simplicity, we implement the same system design and architecture for all problems. Since the output dimension is different for each one, we choose the free parameter V (cf. (3)) to match the output dimension. The functions $\{\Gamma_0, \ldots, \Gamma_G\}$ of the encoding polynomial e are fully connected DNNs with $L = 2$ hidden layers, each with $N = 100$ nodes, and the same non-linear activation. For the computation polynomial h, the DNN Λ_0 has the same architecture as Γ_0, and $\{\Lambda_1, \ldots, \Lambda_P\}$ are linear weight matrices.

As described in Section III, we generate the training datasets randomly for the given problem at every training iteration. This prevents overfitting of the system to a specific dataset. A single step of training consists of a batch of sets $\mathcal{X}$, each with K i.i.d. matrices. Each training epoch includes multiple batches. The scalar values $\{\beta_1, \ldots, \beta_K\}$ are fixed to $\beta_k = k/K$, for $k = 1, \ldots, K$. We implement the training using Tensorflow's (version 2.8.0) Keras module [22].

In the testing phase, we load the model with the learned weights of DNNs corresponding to the training epoch with minimum loss. Besides, we generate the test input data in the same way as for training, except that R (recovery threshold) scalars $\{\alpha_1, \ldots, \alpha_R\}$ are chosen as distinct, but otherwise arbitrary real numbers. In simulations, we obtain these values as $\alpha_n = n/(R + 1)$. The master first obtains the coefficients $\mathbf{U}_0, \ldots, \mathbf{U}_G$ of the encoder, and $\mathbf{V}_0$ of the computation polynomials using the input dataset $\mathcal{X}$. Each encoded matrix $e(\alpha_r), r = 1, \ldots, R$ is communicated to a worker, along with the coefficient $\mathbf{V}_0$. To imitate the erasure effect of the channel, we only use the outputs of computation over these R encoded matrices (corresponding to the recovery threshold). In the final step, the master interpolates the polynomial $v = h \circ e$ using R worker results, and approximates the desired results as $\hat{\boldsymbol{f}}_k \triangleq v(\beta_k) \approx \boldsymbol{f}_k$ for $k = 1, \ldots, K$. Error in function approximation is evaluated in terms of the normalized root mean square error (NRMSE), given by:

$$\frac{\|\hat{f}_k - f(\mathbf{X}_k)\|}{\|f(\mathbf{X}_k)\|}. \tag{6}$$

Simulations are performed on a *high performance computing* (HPC) cluster with nodes equipped with *graphical processing units* (GPUs). A GPU, compared to a *central processing unit* (CPU), is more efficient in parallel processing of large amount data due to its highly parallelized structure. Every HPC cluster node is equipped with NVIDIA Tesla V100 SXM2 GPU with 32Gb RAM, and Intel Xeon(R) Gold 6150 CPU @ 2.70 GHz with 512Gb RAM. The script used for simulations is made available for reproducibility.[1]

C. Results and discussion

The errors in approximate coded computation of the problems $P_1, \ldots, P_4$ are listed in Table I below. The results

---

[1]*https://github.com/navya-xx/AICC.git*

presented here are evaluated on matrices of dimension $M = 50$ with $K = 3$ number of inputs in each dataset and recovery threshold $R = 5$. This shows the ability of the AICC scheme to evaluate these functions approximately in a distributed manner through coded computation.

TABLE I: Function approximation error for AICC.

Problem →	eigenvalue	eigvector	exponential	determinant
NMRSE (%)	4.64%	5.81%	7.85%	1.50%

Figure 4 shows the results for problems P_1 and P_2 in terms of NRMSE for input matrices of dimensions M varying from 50 to 400. The approximation error is low ($< 5\%$), and it is interesting to observe that the error decreases with increasing dimensions M, offering a reasonable accuracy for high dimensional problems. Note that, with the present structure, the number of learnable parameters in the system grows at least at $\mathcal{O}(KM^2)$, whereas the output dimension increases only at $\mathcal{O}(KM)$. This increases the system's capability to learn high-dimensional problems with even higher accuracy, but comes at the price of a longer and more complex training process.

In Figure 5, we compare the runtime of the proposed AICC scheme with standard implementations in Python (called STD for brevity) of the problem P_1 for exact computation of eigenvalues of a matrix. The input dataset consists of 128 batches of $K = 3$ matrices, and runtime is averaged over 1000 runs. The computation task corresponding to each scheme is performed on a single node in the cluster. The runtime of STD is reported for both CPU and GPU based implementations, using the Numpy function `np.linalg.eigvalsh` and the Tensorflow function `tf.linalg.eigvalsh`, respectively. The standard implementation of eigen-problem involves sequential computations [23, Ch.8], and hence, the CPU-based implementation is faster than GPU because it allows quick read/write operations on the memory. The AICC scheme runs all master and worker computations on the same machine, so that it is comparable to the STD computations in terms of computational resources used per unit of time. As M grows, the runtime of the STD scheme increase much faster than that of AICC.

In Figure 6, we compare the performance of P_1 and P_4 with an increasing number of inputs K, while keeping the recovery threshold of the scheme constant at $R = 5$ and the matrix dimension constant at $M = 10$. Notice that the approximation error does not increase too quickly with increasing K. Therefore, for applications with a large number of input matrices needing to be processed at a time, AICC enables approximate recovery with only a small number of worker results.

In further experiments performed over matrix dimensions $M = 10$ and 50, increasing the recovery threshold R showed a slight downward trend in approximation error for small R as expected, but reaching an error floor quickly. Further investigation is required in this regard, but one possible explanation is that the learning saturates for the analyzed problems too soon to show the expected trend.
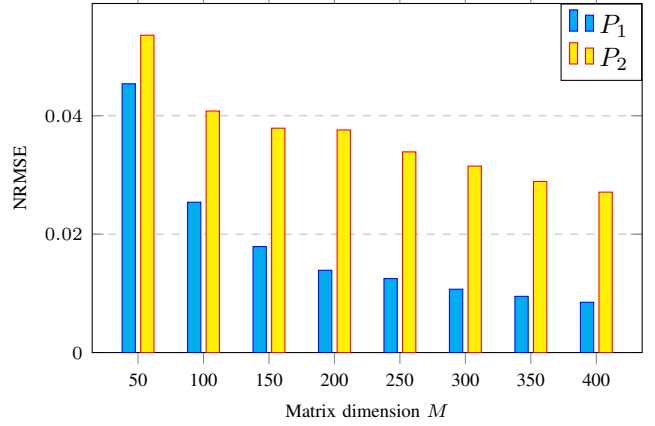

Fig. 4: Approximation error for problems P_1 and P_2 for different matrix dimensions M.
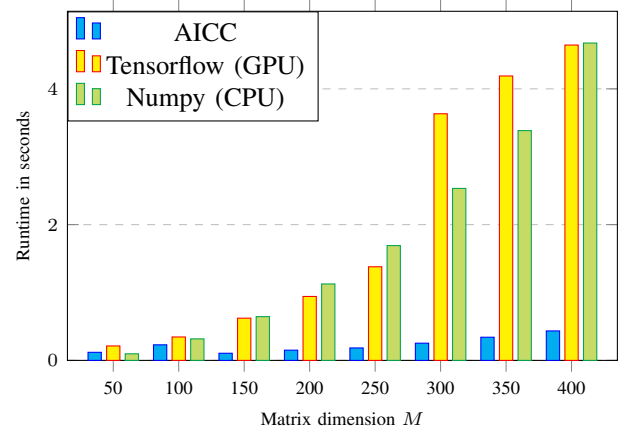

Fig. 5: Comparison of runtime of AICC and standard implementations for computing eigenvalues.
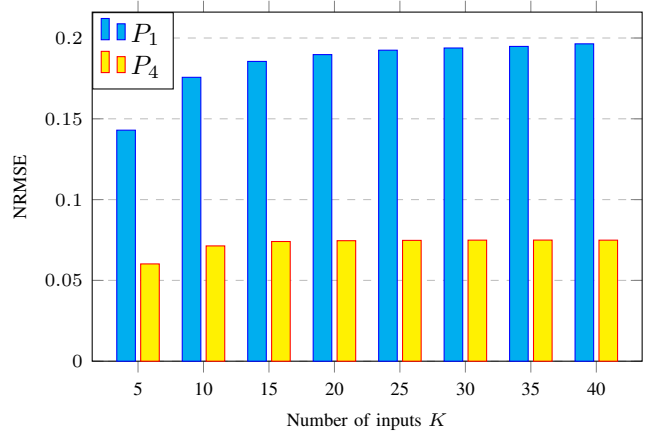

Fig. 6: Varying number of inputs K at fixed recovery threshold $R = 5$.

REFERENCES

[1] C. Li, H. Shen, and T. Huang, "Learning to diagnose stragglers in distributed computing," in *2016 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS)*, 2016, pp. 1–6.

[2] Q. Yu, M. Maddah-Ali, and S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[3] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *IEEE Transactions on Information Theory*, vol. 66, no. 1, pp. 278–301, 2019.

[4] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1215–1225.

[5] M. Soleymani, H. Mahdavifar, and A. S. Avestimehr, "Analog lagrange coded computing," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 1, pp. 283–295, 2021.

[6] W.-T. Chang and R. Tandon, "Random sampling for distributed coded matrix multiplication," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 8187–8191.

[7] N. Charalambides, M. Pilanci, and A. O. Hero, "Approximate weighted cr coded matrix multiplication," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 5095–5099.

[8] T. Jahani-Nezhad and M. A. Maddah-Ali, "Codedsketch: A coding scheme for distributed computation of approximated matrix multiplication," *IEEE Transactions on Information Theory*, vol. 67, no. 6, pp. 4185–4196, 2021.

[9] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, "Oversketch: Approximate matrix multiplication for the cloud," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 298–304.

[10] S. Kiani and S. C. Draper, "Successive approximation coding for distributed matrix multiplication," *arXiv preprint arXiv:2201.03486*, 2022.

[11] H. Jeong, A. Devulapalli, V. R. Cadambe, and F. P. Calmon, "ϵ-approximate coded matrix multiplication is nearly twice as efficient as exact multiplication," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 3, pp. 845–854, 2021.

[12] L. Chen, H. Wang, Z. Charles, and D. Papailiopoulos, "Draco: Byzantine-resilient distributed training via redundant gradients," in *International Conference on Machine Learning*. PMLR, 2018, pp. 903–912.

[13] M. M. Amiri and D. Gündüz, "Computation scheduling for distributed machine learning with straggling workers," *IEEE Transactions on Signal Processing*, vol. 67, no. 24, pp. 6270–6284, 2019.

[14] J. Kosaian, K. Rashmi, and S. Venkataraman, "Learning a code: Machine learning for approximate non-linear coded computation," *arXiv preprint arXiv:1806.01259*, 2018.

[15] ——, "Parity models: erasure-coded resilience for prediction serving systems," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 30–46.

[16] ——, "Learning-based coded computation," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 227–236, 2020.

[17] L. Miretti, R. L. Cavalcante, S. Stanczak, M. Schubert, R. Boehnke, and W. Xu, "Closed-form max-min power control for some cellular and cell-free massive mimo networks," *arXiv preprint arXiv:2205.00429*, 2022.

[18] A. Alkhateeb and R. W. Heath, "Frequency selective hybrid precoding for limited feedback millimeter wave systems," *IEEE Transactions on Communications*, vol. 64, no. 5, pp. 1801–1818, 2016.

[19] A. H. Al-Mohy and N. J. Higham, "A new scaling and squaring algorithm for the matrix exponential," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 970–989, 2010.

[20] A. M. Ostrowski, *Sur l'approximation du determinant de Fredholm par les determinants des systèmes d'équations linéaires*. Almqvist & Wiksell, 1938.

[21] R. P. Brent, J. anne H. Osborn, and W. D. Smith, "Note on best possible bounds for determinants of matrices close to the identity matrix," *Linear Algebra and its Applications*, vol. 466, pp. 21–26, 2015.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[23] B. N. Parlett, *The symmetric eigenvalue problem*. SIAM, 1998.