This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAIT.2020.2983165, IEEE Journal on Selected Areas in Information Theory

Learning-Based Coded Computation

Jack Kosaian, K.V. Rashmi, and Shivaram Venkataraman

***Abstract*—Recent advances have shown the potential for coded computation to impart resilience against slowdowns and failures that occur in distributed computing systems. However, existing coded computation approaches are either unable to support non-linear computations, or can only support a limited subset of non-linear computations while requiring high resource overhead. In this work, we propose a *learning-based* coded computation framework to overcome the challenges of performing coded computation for general non-linear functions. We show that careful use of machine learning within the coded computation framework can extend the reach of coded computation to imparting resilience to more general non-linear computations. We showcase the applicability of learning-based coded computation to neural network inference, a major workload in production services. Our evaluation results show that learning-based coded computation enables accurate reconstruction of unavailable results from widely deployed neural networks for a variety of inference tasks such as image classification, speech recognition, and object localization. We implement our proposed approach atop an open-source prediction serving system and show its promise in alleviating slowdowns that occur in neural network inference. These results indicate the potential for learning-based approaches to open new doors for the use of coded computation for broader, non-linear computations.**

I. Introduction

LARGE-scale production services increasingly depend on distributed computation, in which execution is performed on many servers. It is well known that common distributed computing environments are prone to unavailabilities (e.g., slowdowns and failures) that can inflate tail latency, extend job completion times, and cause violations of latency agreements [1]. Coded computation has recently been revitalized as a potential approach to alleviate the effects of slowdowns and failures in such systems while using fewer resources than replication-based approaches [2], [3]. Under coded computation, one encodes the inputs to computation and performs computation over encoded inputs to impart resilience against unavailabilities. There have been many recent advances in coded computation that have shown its applicability to distributed computations such as matrix multiplication and convolution, among others [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14].

Despite these recent advances in coded computation, most existing work focuses on *linear* computations. This significantly limits the applications to which coded computation may be applied. For example, many popular machine learning models, such as neural networks, are complex non-linear functions. A recently proposed class of codes [13] extends the use of coded computation to multivariate polynomial functions, but requires as much or more resource overhead than replication-based approaches. Thus, while coded computation is a promising technique for mitigating slowdowns and failures, existing approaches are insufficient for imparting resilience to broader classes of non-linear computations.

Machine learning has recently led to significant advances in complex tasks, such as image classification and natural language processing. This leads one to question: *can machine learning similarly help overcome the challenges of coded computation for non-linear functions?*

In this work, we answer this question in the affirmative by proposing and evaluating a *learning-based* coded computation framework. We describe two distinct paradigms for leveraging machine learning for coded computation: (1) *Learning a code:* using neural networks as encoders and decoders to learn a code that enables coded computation over non-linear functions. (2) *Learning a parity computation:*[1] using simple encoders and decoders (e.g., addition/subtraction), and instead learning a new computation over parities that enables reconstruction of unavailable outputs. These two methods are fundamentally new approaches to coded computation.

The techniques that we develop have the potential for applicability to a broad class of computations. For concreteness, we focus on imparting resilience to machine learning models during inference, specifically, neural networks. Inference is the process of using a trained machine learning model to produce predictions in response to input queries. Large-scale services typically perform inference in so-called "prediction serving systems" in which multiple servers run a copy of the same machine learning model and queries are load-balanced across these servers. We focus on inference because it is commonly deployed in latency-sensitive services in which slowdowns and failures can jeopardize user-

J. Kosaian and K.V. Rashmi are with the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 15213 USA e-mail: jkosaian@cs.cmu.edu, rvinayak@cs.cmu.edu.

S. Venkataraman is with the Computer Science Department, University of Wisconsin, Madison, WI, 53715 USA e-mail: shivaram@cs.wisc.edu

[1]This approach appears in part in our ACM SOSP 2019 paper [15].

experience [16]. Neural network inference also represents a challenging non-linear computation for which previous coded computation approaches are inapplicable. While neural networks do contain linear components (e.g., matrix multiplication), they also contain many non-linear components (e.g., activation functions, max-pooling), which make the overall function computed by the neural network non-linear.

Using machine learning for coded computation leads to the reconstruction of *approximations* of unavailable results of computation. This is appropriate for imparting resilience to inference, as the results of inference are themselves approximate. Furthermore, any inaccuracy incurred due to employing learning only comes into play when a result from inference would otherwise be slow or failed. In this case, many services prefer a slightly less accurate result as compared to a late one [16].

We show that learning-based coded computation enables accurate reconstruction of unavailable predictions for a range of neural network architectures and inference tasks, including image classification, speech recognition, and object localization. For example, using half of the additional resources of a replication-based approach, learning-based coded computation reconstructs unavailable predictions from a ResNet-18 model to be within a 6.5% difference in accuracy compared to if the original predictions were not slow or failed. We additionally implement learning-based coded computation atop an open-source prediction serving system, and show its ability to reduce tail latency by up to 48%. These results highlight the potential of learning-based approaches to open new doors to the use of coded computation for broader classes of non-linear computations.

Our implementations of the proposed approaches and evaluation setup are made available on Github at (1) https://github.com/Thesys-lab/learned-cc and (2) https://github.com/Thesys-lab/parity-models.

II. Related Work

Section I briefly discussed the large body of work on coded computation for linear computations, and the fact that current approaches to coded computation for restricted classes of non-linear computations are resource intensive. We now discuss other related work.

A recent approach [17] performs coded computation over the linear operations of neural networks and decodes before each non-linearity. This requires splitting the operations of a neural network onto multiple servers and many decoding steps, which may increase latency even in the absence of slowdown or failure. In contrast, our approaches perform coded computation over a neural network as a whole with a single decoding step.

A related approach [18] builds on top of our initial proposal of learning-based coded computation for inference [19]. This related approach [18] focuses specifically on image classification and proposes concatenating multiple images for inference. In contrast, we propose a *general* framework for learning-based coded computation; we show that our proposed framework is applicable to a variety of inference tasks, including image classification, speech recognition, and object localization. The technique described in this related approach [18], in fact, fits within our proposed framework as an example of an encoder specialized for image classification, similar to the concatenation-based encoder that will be described in Section V-A, and is evaluated in our prior work [15].

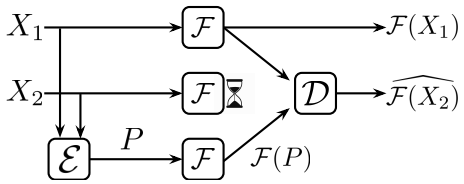


Fig. 1: Example of coded computation with $k = 2$ original units and $r = 1$ parity units.

Several coded computation approaches have been proposed for gradient computation in a straggler-resistant manner when *training* a machine learning model [12], [20], [21], [22]. In contrast to these works, our focus is on the *inference* phase of machine learning.

Finally, a number of recent works have explored learning error correcting codes for communication [23], [24], [25]. The goal of these works is to recover data units transmitted over a noisy channel. In contrast, our goal is to recover the outputs of computation over data units. To the best of our knowledge, we present the first approach that leverages learning for coded computation.

III. Setting, Notation, and Metrics

This section describes the coded computation setting, notation, and metrics used in this work.

A. Setting and Notation

We consider a setting in which k copies of a computation $\mathcal{F}$ are performed on separate servers. Each input X_i, is sent to one of the copies of $\mathcal{F}$ to compute and return $\mathcal{F}(X_i)$. Thus, given k inputs $X_1, X_2, \ldots, X_k$, the goal is to compute $\mathcal{F}(X_1), \mathcal{F}(X_2), \ldots, \mathcal{F}(X_k)$. As depicted in Fig. 1, coded computation introduces to this system an encoder $\mathcal{E}$, a decoder $\mathcal{D}$, and r additional copies of $\mathcal{F}$. The encoder $\mathcal{E}$ takes in k original inputs $X_1, X_2, \ldots, X_k$ and produces r parities $P_1, P_2, \ldots, P_r$. All original and parity inputs are sent to copies of $\mathcal{F}$. Given any k out of the total $(k + r)$ original and parity outputs from copies of $\mathcal{F}$, the decoder $\mathcal{D}$ reconstructs the original k outputs $\mathcal{F}(X_1), \mathcal{F}(X_2), \ldots, \mathcal{F}(X_k)$. We denote a reconstructed output for input X_i as $\widehat{\mathcal{F}(X_i)}$. This setup can recover from up to r slow or failed computations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAIT.2020.2983165, IEEE Journal on Selected Areas in Information Theory

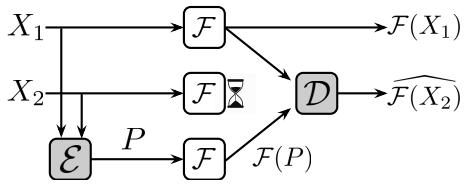


Fig. 2: Example of coded computation using learned encoders and decoders (darkly shaded boxes).

Throughout this paper, we focus on the scenario in which $r = 1$. This represents the typical unavailability faced by groups of $(k+r)$ servers ("stripes") for typical values of k and r in datacenters [26]. Section VI-B describes how the approaches proposed in this paper can be extended to support larger values of r.

Within the context of neural network inference, each function $\mathcal{F}$ is a neural network. Inputs X_i are queries issued from applications (e.g., images), and the output of computation $\mathcal{F}(X_i)$ is a prediction resulting from inference. When describing coded computation in the setting of inference, we refer to $\mathcal{F}$ as the "base model."

B. Metrics

As described in Section I, we consider two different approaches to learning-based coded computation, which will be described and evaluated separately. We now describe the accuracy metrics used throughout evaluation.

Analyzing erasure codes for traditional applications such as storage and communication involves reasoning about performance under normal operation (when unavailability does not occur) and performance in "degraded mode" (when unavailability occurs and reconstruction is required). These modes of operation are similarly present for learning-based coded computation.

The overall accuracy of an inference system is based on its accuracy when base model outputs are available (A_a) and its accuracy when these outputs are unavailable (A_d, "degraded mode"). If f fraction of base model outputs are unavailable, the overall accuracy (A_o) is:

$$A_o = (1 - f)A_a + fA_d \tag{1}$$

The goal in learning-based coded computation is to achieve high A_d; the approaches proposed in this paper do not change accuracy when the original function outputs are available (A_a).

We focus our evaluation on recovering unavailable predictions resulting from neural network inference. All reported results use test datasets, which are not used in training. Test samples are randomly placed into groups of k and encoded to produce a parity. We then perform decoding for each of the k different scenarios in which one output is unavailable. Each result from decoding is compared to the true label for its corresponding input.

IV. Learning Encoders and Decoders

In this section, we describe our first approach for learning-based coded computation: *learning erasure codes*. Recall that the coded computation setup described in Section III-A and illustrated in Fig. 1 has three components: the given function $\mathcal{F}$, the encoder $\mathcal{E}$, and the decoder $\mathcal{D}$. The goal in this section is to learn an encoder $\mathcal{E}$ and a decoder $\mathcal{D}$ that accurately reconstruct unavailable outputs from function $\mathcal{F}$. We use neural networks to learn encoders and decoders due to their recent success in a number of tasks. Fig. 2 displays the differences between this approach and the traditional coded computation framework.

We next describe how encoders and decoders are trained, and subsequently describe the neural network architectures for learning encoders and decoders.

A. Training encoders and decoders

The goal of training is to learn neural network encoders and the decoders that accurately reconstruct unavailable function outputs. The given function $\mathcal{F}$ is not modified during training.

When the given function $\mathcal{F}$ is a machine learning model, the encoder and the decoder are trained using the same training dataset that was used to train $\mathcal{F}$. When such a dataset is not available, which will be the case for generic functions $\mathcal{F}$ outside the realm of machine learning, one can instead generate a training dataset comprising pairs $(X, \mathcal{F}(X))$ for values of X in the domain of $\mathcal{F}$. Each sample for training the encoder and decoder uses a set of k (randomly chosen) inputs from the training dataset. A forward and backward pass is performed for each of the $\binom{k+r}{r}$ possible unavailability scenarios, except for the case where all unavailable outputs correspond to parity inputs. An iterative optimization algorithm, such as gradient descent, updates the encoder and decoder's parameters during training.

A forward and a backward pass under this training setup is illustrated in Fig. 3. During a forward pass, the k data inputs $X_1, X_2, \ldots, X_k$ are fed through the encoder to generate r parity inputs $P_1, P_2, \ldots, P_r$. Each of the $(k + r)$ inputs (data and parity) are then fed through the given function $\mathcal{F}$. The resulting $(k + r)$ outputs $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k), \mathcal{F}(P_1), \ldots, \mathcal{F}(P_r)$ are fed through the decoder $\mathcal{D}$, and up to r of these outputs are made unavailable (detailed in Section IV-C1). The decoder outputs (approximate) reconstructions for the unavailable function outputs among $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$. A backward pass involves using any chosen loss function (detailed below) and backpropogating through $\mathcal{D}$, $\mathcal{F}$, and $\mathcal{E}$. We train the encoder and decoder in tandem by backpropagating through $\mathcal{F}$, making this approach applicable to any numerical differentiable function $\mathcal{F}$.

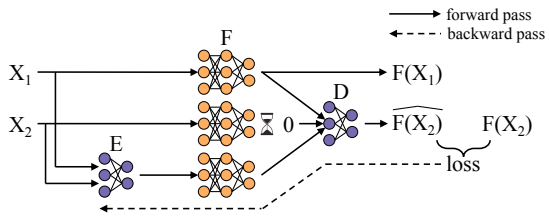


Fig. 3: A forward and a backward pass in training the encoder and decoder with $k = 2$ and $r = 1$.

Layer	MLPEncoder	ConvEncoder
1	FC: $kn^2 \times kn^2$	Conv: 3×3, dilation 1
2	FC: $kn^2 \times rn^2$	Conv: 3×3, dilation 1
3		Conv: 3×3, dilation 2
4		Conv: 3×3, dilation 4
5		Conv: 3×3, dilation 8
6		Conv: 3×3, dilation 1
7		Conv: 1×1, dilation 1

TABLE I: Neural network architectures for encoders using fully-connected (FC) and convolutional (Conv) layers. All convolutional layers have stride of 1. In each network, ReLUs follow all but the final layer.

Many loss functions can be used in training encoders and decoders. For simplicity, we use as a loss function the mean-squared error between the reconstructed output $\widehat{\mathcal{F}(X)}$ and the output $\mathcal{F}(X)$ that would be returned if the base model were not slow or failed. This loss function is general enough to be applicable to many functions $\mathcal{F}$. A loss function that is specific to $\mathcal{F}$ may also be used, such as cross-entropy loss for image classification tasks.

We next describe the encoder and decoder architectures used in this approach. For concreteness, we describe the proposed architectures by setting the given function $\mathcal{F}$ as a neural network image classifier over m classes. For such an $\mathcal{F}$, each data input X is an $n \times n$ pixel image. Each output $\mathcal{F}(X)$ is an m-length vector resulting from the last layer of the neural network $\mathcal{F}$.

B. Encoder architectures

We consider two neural network architectures for learning the encoder.

1) MLPEncoder: We first consider a simple 2-layer multilayer perceptron (MLP) encoder architecture, which we call *MLPEncoder*. Under this architecture, each $n \times n$ data input is flattened into an n^2-length vector, as illustrated in Fig. 4a. The k flattened vectors from inputs $X_1, X_2, \ldots, X_k$, are concatenated to form a single kn^2-length input vector to the MLP. The first fully-connected layer of the MLP produces a kn^2-length hidden vector. The second fully-connected layer produces an rn^2-length output vector, which represents the r parity inputs. Each layer used in MLPEncoder is outlined in Table I.

Layer	Decoder
1	FC: $(k + r)m \times km$
2	FC: $km \times km$
3	FC: $km \times km$

TABLE II: Neural network architecture for decoder employing fully-connected (FC) layers. ReLU activation functions are used after all but the final layer.

The fully-connected nature of the MLP allows for computation of arbitrary combinations of the kn^2 input features using few layers. While effective for many scenarios (as will be shown in Section IV-D), the high parameter count of the fully-connected layers can lead to overfitting. We next describe an alternate encoder architecture that is less susceptible to overfitting.

2) ConvEncoder: The ConvEncoder architecture makes use of convolutional layers instead of fully-connected layers. Unlike MLPEncoder, ConvEncoder operates over data inputs in their original two-dimensional $n \times n$ representation. As depicted in Fig. 4b, the k inputs to the encoder are treated as k *input channels* to the first convolution layer. This is similar to the representations of RGB images for convolutional neural networks in image classification. We explain how the encoder handles multi-channel inputs in Section IV-B3.

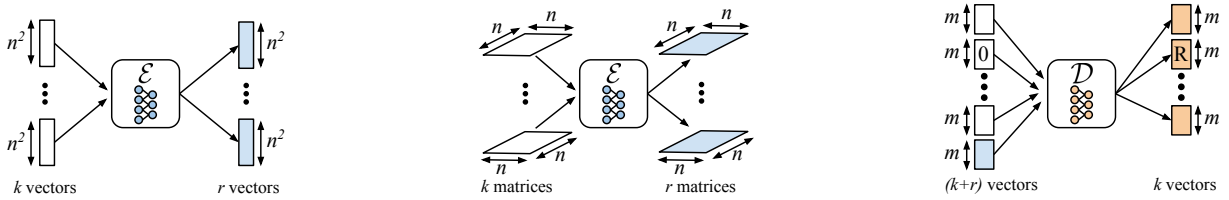
Convolutional neural networks used for image classification repeatedly downsample an image to expand the receptive field of convolutional filters. This works well when the output of the network is much smaller than the input, which is typically the case in image classification. However, an encoder produces outputs that are the same size as the inputs, as shown in Fig. 4b. Hence, using convolutional layers with downsampling would necessitate subsequent upsampling to bring the outputs back to the input size. This has been shown to be inefficient for image segmentation tasks [27]. We overcome this issue by using dilated convolutions, which increase the receptive field of convolutional filters exponentially with linear increase in the number of layers [27].

Table I shows each layer of ConvEncoder. The first layer has k input channels and the final layer has r output channels, one for each parity to be produced. Each intermediate layer has $20k$ input channels and $20k$ output channels. We increase the receptive field of convolutions by increasing the dilation factor, borrowing this architecture from its successful use in image segmentation [27].

By employing convolutions in place of fully-connected layers, ConvEncoder has less parameters than MLPEncoder but requires more layers to enable combinations of all input pixels. This lower parameter count helps avoid overfitting, as will be shown in Section IV-D.

3) Multi-channel input: It is common to represent color images as having multiple channels. For example,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAIT.2020.2983165, IEEE Journal on Selected Areas in Information Theory



(a) MLPEncoder **(b)** ConvEncoder **(c)** Decoder

Fig. 4: Inputs and outputs of each encoder and decoder. In this decoding example, the second input is unavailable. The second output represents its reconstruction (marked with "R"). Parities are shaded in blue.

an RGB image consists of 3 channels representing the pixel values of an image's red, green, and blue components. The encoders described above handle multi-channel inputs by encoding across each channel independently. An encoder with k RGB images as input would encode across the k red channels to produce r "red" parity channels, and similarly for green and blue channels. The r "red," "green," and "blue" parity channels are combined to create r parity "images."

C. Decoder architecture

Fig. 4c shows a high-level overview of the decoder. The two key design choices for the decoder architecture are: (1) the representation of the unavailable base model outputs at the input layer of the decoder, and (2) the neural network architecture used for the decoder.

1) Representing unavailability: The decoder takes as input the $(k + r)$ vectors of length m, $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k), \mathcal{F}(P_1), \ldots, \mathcal{F}(P_r)$, where $X_1, \ldots, X_k$ are function inputs and $P_1, \ldots, P_r$ are parities generated by the encoder. Some of these inputs to the decoder could be unavailable. In place of any unavailable input, we insert a vector of all zeros. If a vector of all zeros is a valid output for a given $\mathcal{F}$, an alternative value may be used to represent unavailability.

An alternative approach is to provide the decoder with only the (concatenated) available inputs. We chose the former as it allows the decoder to use the positions of unavailable inputs as features when learning. Providing only the available inputs to the decoder would hide this information. This choice is inspired by traditional (handcrafted) decoders, which typically leverage positional information. Correspondingly, the output of the decoder consists of k vectors $\widehat{\mathcal{F}(X_1)}, \ldots, \widehat{\mathcal{F}(X_k)}$, each representing an approximate reconstruction of the corresponding potentially unavailable function output.

2) Decoder architecture: We use a 3-layer MLP architecture for the decoder, as shown in Table II. The raw outputs of the base model $\mathcal{F}$ are inputs to the decoder. These outputs are not converted to a probability distribution, as is typically done for neural network classifiers via a softmax operation, because the output for a given function $\mathcal{F}$ may not necessarily be constrained to be a probability distribution.

Base Model	MNIST	Fashion-MNIST	CIFAR-10
ResNet-18	99.20	92.85	93.47
Base-MLP	97.93	89.47	-

TABLE III: Test accuracies for evaluated base models.

D. Evaluation

We evaluate the accuracy of learned encoders and decoders by focusing on the scenario in which $\mathcal{F}$ is a neural network used for image classification.

1) Setup: We implement all the encoder and decoder architectures in PyTorch. We evaluate this approach on the MNIST, Fashion-MNIST [28], and CIFAR-10 datasets. As this work represents the first use of learning for coded computation, we use these datasets to establish the potential of learned encoders and decoders.

a) Base models: We use two neural networks as base models $\mathcal{F}$: Base-MLP and ResNet-18. Base-MLP is a 3-layer multilayer perceptron containing three fully-connected layers with dimensions 784×200, 200×100, and 100×10 and ReLUs following all but the final layer. ResNet-18 [29] is a widely used neural network consisting of many convolutional, pooling, and fully-connected layers. We use ResNet-18 for two reasons: (1) it achieves high accuracy on many tasks, and (2) it is a significantly more complex model than Base-MLP and thus provides an alternative evaluation point for our proposed approach. Table III shows the "available" accuracies of the base models, as described in Section III. We do not use Base-MLP for CIFAR-10, as similar architectures achieve low accuracy on this dataset [30].

b) Hyperparameters: We perform experiments for all datasets and base models listed above for $k = 2$ with $r = 1$. Training ueses minibatches of 64 samples. Each sample in the minibatch consists of k images drawn randomly without replacement; no image is sampled more than once per epoch. The encoder and decoder are

Dataset	Base Model	Encoder	Degraded Acc.
MNIST	Base-MLP	MLP	97.76
		Conv.	97.70
	ResNet-18	MLP	98.06
		Conv.	98.88
Fashion	Base-MLP	MLP	81.96
		Conv.	82.53
	ResNet-18	MLP	88.15
		Conv.	88.92
CIFAR-10	ResNet-18	MLP	41.16
		Conv.	82.04

TABLE IV: Degraded mode accuracy for each configuration considered, with $k = 2$ and $r = 1$.

trained in tandem using the Adam optimizer [31] with learning rate of 0.001 and L2-regularization of 10^{-5}. The weights for the convolutional layers are initialized via uniform Xavier initialization [32], weights for the fully-connected layer are initialized according to $\mathcal{N}(0, 0.01)$, and bias values are initialized to zero.

2) Results: Table IV presents evaluation results on test datasets for all combinations of datasets and base models with $k = 2$ and $r = 1$. Across all datasets and configurations, the best of the proposed learned encoder and decoder pair achieve a degraded mode accuracy of no more than 11.5% lower than available accuracy. This is a small drop in accuracy compared to the available mode accuracies shown in Table III when considering that reconstructions only come into play when a function output would otherwise be slow or failed. This represents a promising step forward in coded computation for neural networks, as existing coded computation approaches are inapplicable even for simple neural networks.

a) Complexity of the base model: We find that the complexity of the base model does not have an adverse effect on the accuracy of the learned code. Despite the higher complexity of ResNet-18 than Base-MLP, the learned code achieves similar accuracies for both models. This suggests that the proposed approach is effective even for complex base models.

b) Encoder architectures: For the MNIST and Fashion-MNIST datasets, there is little difference in the accuracies attained by the two proposed neural network encoding function architectures. The difference between the two architectures comes to fore in the more complex CIFAR-10 dataset, where ConvEncoder greatly outperforms MLPEncoder. MLPEncoder's high parameter count causes it to overfit on CIFAR-10, while ConvEncoder is able to reach higher accuracy.

E. Practical considerations

While the results presented above show the promise of learning encoders and decoders for coded computation, there are several practical challenges with deploying this approach in prediction serving systems. Many prediction serving systems contain a frontend node that receives queries and load-balances them across backend nodes that perform inference. Within this setup, the frontend is the logical place to perform encoding and decoding operations; the frontend encodes k queries to generate a parity, and decodes slow or failed predictions after receiving k outputs. As neural networks are often computationally expensive, using neural network encoders and decoders would increase the latency of recovering from a slowdown or failure. This approach necessitates using hardware acceleration for encoders and decoders, which requires using a more expensive frontend node.

We next describe an alternate learning-based approach aimed at alleviating these practical challenges.

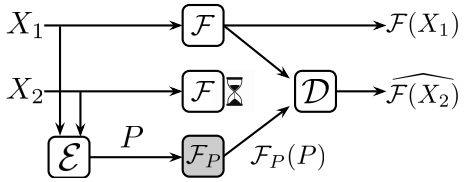


Fig. 5: Example of coded computation using a parity model (darkly shaded box).

V. Learning a Parity Computation

To overcome the practical concerns of learned encoders and decoders, we propose a fundamentally new approach to coded computation. Rather than designing new encoders and decoders, we propose to use simple, fast encoders and decoders (such as addition and subtraction) and instead design a *new computation over parities*. Within the context of machine learning inference, this new computation is a separate model, which we call a "parity model." As depicted in Fig. 5, instead of the extra copy of $\mathcal{F}$ deployed by current coded computation approaches (see Fig. 1), we introduce a parity model, which we denote as $\mathcal{F}_P$. The challenge of this approach is to design a parity model that enables accurate reconstruction of unavailable function outputs. We address this by designing parity models as neural networks, and learning a parity model that enables simple encoders and decoders to reconstruct slow or failed function outputs.

By learning a parity model and using simple, fast encoders and decoders, this approach is able to impart resilience to non-linear computations, like neural networks, while operating with low latency without requiring hardware acceleration for encoding and decoding.

A. Encoders and decoders

Introducing and learning parity models enables the use of many different encoder and decoder designs,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAIT.2020.2983165, IEEE Journal on Selected Areas in Information Theory

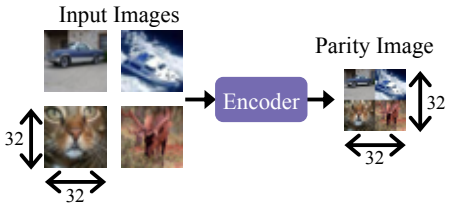


Fig. 6: Example of an image-specific encoder that downsamples and concatenates images into a parity image.

opening up a rich design space. In this paper, we illustrate the potential of parity models by using a simple addition/subtraction erasure code. Even with this simple encoder and decoder, we show that parity models can still accurately reconstruct unavailable function outputs. This simple encoder and decoder is applicable to a wide range of inference tasks, including image classification, speech recognition, and object localization. A system that is specialized for one inference task may benefit from designing task-specific encoders and decoders for use within this framework, such as an encoder that downsamples and concatenates images for image classification as depicted in Fig. 6. Our prior work has shown the efficacy of such task-specific encoders [15].

Under this simple addition/subtraction encoder and decoder, the encoder produces a parity as the summation of k inputs, i.e., $P = \sum_{i=1}^{k} X_i$. Inputs are normalized to a common size prior to encoding, and summation is performed across corresponding features of each input (e.g., top-right pixel of each image). The decoder subtracts $(k-1)$ available function outputs from the output of the parity model $\mathcal{F}_P(P)$ to reconstruct an unavailable output. Thus, an unavailable output $\mathcal{F}(X_j)$ is reconstructed as $\widehat{\mathcal{F}(X_j)} = \mathcal{F}_P(P) - \sum_{i \neq j}^{k} \mathcal{F}(X_i)$.

B. Parity model design

We use neural networks for parity models to learn a model that transforms parities into a form that enables decoding. For a parity model to help in mitigating slowdowns, the average runtime of a parity model should be similar to that of the base model. One way of enforcing this is by using the same neural network architecture for the parity model as is used for the base model (i.e., same number and size of layers). Thus, if the base model is a ResNet-18 architecture, the parity model also uses ResNet-18, but trained using the procedure that will be described in Section V-C. As a neural network's architecture determines its runtime, this approach ensures that the parity model has the same average runtime as the base model. We use this approach in our evaluation.

In general, a parity model is not required to use the same architecture as the base model. In cases where it is necessary or preferable to use a different architecture, such as when $\mathcal{F}$ is not a neural network, a parity model could potentially be designed via neural architecture search [33]. However, we do not focus on this scenario.

We do not apply a softmax to the output of a parity model, as the desired output of a parity model is not necessarily a probability distribution.

C. Training a parity model

1) Training data: The training data for a parity model are the parities generated by the encoder, and training labels are the outputs expected by the decoder. For the simple encoder and decoder described in Section V-A, with $k = 2$, training data from inputs X_1 and X_2 are $(X_1 + X_2)$ and labels are $(\mathcal{F}(X_1) + \mathcal{F}(X_2))$.

Training data is generated using inputs that are representative of those issued to the base model for inference. A parity model is trained using the same dataset used for training the base model, whenever available. Thus, if the base model was trained using the CIFAR-10 dataset, samples from CIFAR-10 are used as inputs $X_1, \ldots, X_k$ that are encoded together to generate training samples for the parity model. The desired parity model output is generated by performing inference with the base model to obtain $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$ and summing these outputs. For example, if the outputs of a base model are vectors of n floating point numbers, as is the case in classification with n classes, a label would be the element-wise summation of these vectors. One can also use as labels the summation of the true labels for inputs.

2) Loss function: Many loss functions can be used in training. We use the mean-squared-error between the output of the parity model and the desired output as the loss function. We choose mean-squared-error rather than a task-specific loss function (e.g., cross-entropy) to make this approach applicable to many inference tasks.

3) Training procedure: A parity model is trained prior to being deployed. Training a parity model involves the same iterative optimization procedure commonly used to train neural networks. In each iteration, k samples are drawn at random from the base model's training dataset and encoded to form a parity sample. The parity model performs a forward pass over this parity sample to produce $\mathcal{F}_P(P)$. A loss value is calculated between $\mathcal{F}_P(P)$ and the desired parity model output (e.g., $\mathcal{F}(X_1) + \mathcal{F}(X_2)$ for the addition/subtraction code with $k = 2$). Parity model parameters are updated via the standard backpropogation algorithm. This process continues until a parity model reaches a sufficient accuracy.

D. Example

Fig. 7 shows an example of how parity models mitigate the unavailability of any one of three instances of a base model (i.e., $k = 3$). Inputs X_1, X_2, X_3 are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAIT.2020.2983165, IEEE Journal on Selected Areas in Information Theory

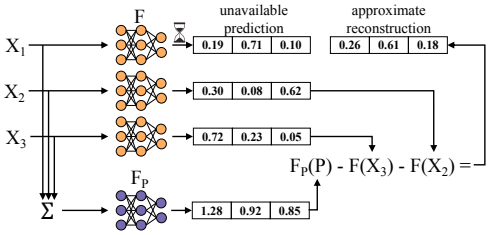


Fig. 7: Example of using a parity model with $k = 3$.

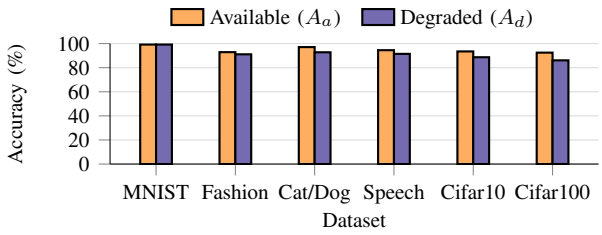


Fig. 8: Comparison of accuracy when predictions from the base model are available (A_a) and when reconstructions via a parity model with $k = 2$ are necessary (A_d).

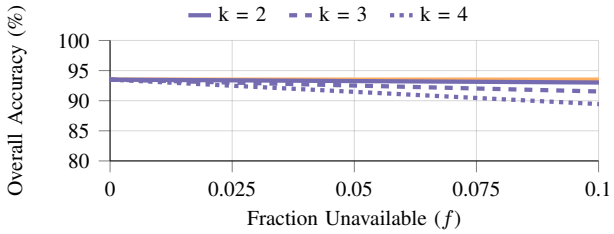


Fig. 9: Overall accuracy (A_o) on CIFAR-10 with varying unavailability. The horizontal orange line shows the accuracy of the ResNet-18 base model (A_a).

dispatched to three model instances for inference on base model $\mathcal{F}$ to return outputs $\mathcal{F}(X_1), \mathcal{F}(X_2), \mathcal{F}(X_3)$. As inputs are dispatched to model instances, they are encoded (Σ) to generate a parity $P = (X_1 + X_2 + X_3)$. The parity is dispatched to a parity model $\mathcal{F}_P$ to produce $\mathcal{F}_P(P)$. In this example, the model instance processing X_1 is slow. The decoder reconstructs this output as $(\mathcal{F}_P(P) - \mathcal{F}(X_3) - \mathcal{F}(X_2))$. The reconstruction in this example provides a reasonable approximation of the unavailable output (labeled "unavailable prediction").

E. Evaluation of accuracy

We now evaluate the accuracy of reconstructing unavailable function outputs with parity models. We again focus on neural network inference, that is, when each $\mathcal{F}$ is a neural network and the goal is to reconstruct unavailable predictions resulting from inference.

1) Setup: We use PyTorch to train parity models for each parameter k, dataset, and base model. We use popular image classification (CIFAR-10 and 100, Cat v. Dog [34], Fashion-MNIST [28], and MNIST), speech recognition (Google Commands [35]), and object localization (CUB-200 [36]) tasks. For CIFAR-100, we report top-5 accuracy, as is common (i.e., the fraction for which the true class of X_i is in the top 5 of $\widehat{\mathcal{F}(X_i)}$).

As described in Section V-B, a parity model uses the same neural network architecture as the base model. We consider five architectures: a multilayer perception (MLP),[2] LeNet-5 [37], VGG-11 [38], ResNet-18, and ResNet-152 [29]. The former two are simpler neural networks while the others are widely-used neural networks.

a) Parameters: We consider values for parameter k of 2, 3, and 4, with $r = 1$, corresponding to 33%, 25%, and 20% redundancy. We use Adam [31], learning rate of 0.001, L2-regularization of 10^{-5}, and batch sizes between 32 and 64. Convolutional layers use Xavier initialization [32], biases are initialized to zero, and other weights are initialized from $\mathcal{N}(0, 0.01)$. We use the generic addition encoder and subtraction decoder described in Section V-A. Our prior work [15] evaluates parity models using other encoders and decoders.

[2]The MLP has two hidden layers with 200 and 100 units and ReLUs.

2) Results: Fig. 8 shows the accuracy of the base model (A_a) and the degraded mode accuracy (A_d) when using parity models with $k = 2$ for image classification and speech recognition tasks. VGG-11 is used for the speech dataset, ResNet-152 for CIFAR-100, and ResNet-18 for all others. The degraded mode accuracy when using parity models is no more than 6.5% lower than that when predictions from the base model are available. As Fig. 9 illustrates, this enables parity models to maintain high overall accuracy (A_o). For example, at expected levels of unavailability (i.e., f less than 10%), the overall accuracy when using parity models on the CIFAR-10 dataset is at most 0.4%, 1.9%, and 4.1% lower than when all predictions are available at k values of 2, 3, and 4, respectively. This indicates a tradeoff between parameter k, which controls resource-efficiency and resilience, and the accuracy of reconstructions, which we discuss below.

a) Neural network architectures: We observe high degrade mode accuracy when using a variety of neural network architectures. For example, on the Fashion-MNIST dataset, degraded mode accuracy for the MLP, LeNet-5, and ResNet-18 models are only 1.7–9.8% lower than when slowdown or failure does not occur.

b) Object localization: We next evaluate parity models on object localization, which is a regression task. The goal in this task is to predict the coordinates of a bounding box surrounding an object of interest in an image. We evaluate the applicability of parity models to this task using the Caltech-UCSD Birds

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAIT.2020.2983165, IEEE Journal on Selected Areas in Information Theory

Fig. 10: Example reconstruction for object localization.

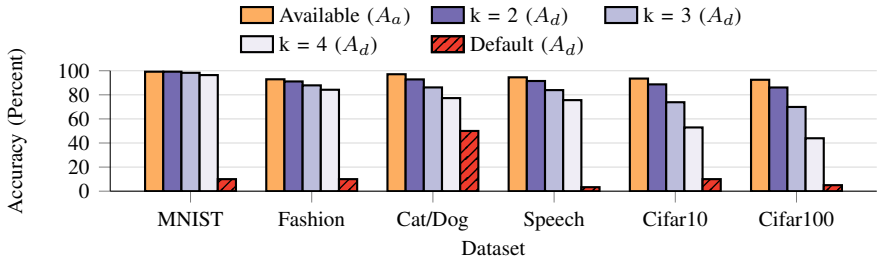


Fig. 11: Accuracies of predictions reconstructed using parity models compared to returning a default response when base model predictions are unavailable (A_d).

dataset [36] with ResNet-18. The performance metric for localization tasks is the intersection over union (IoU): the IoU between two bounding boxes is computed as the area of their intersection divided by the area of their union. IoU values fall between 0 and 1, with an IoU of 1 corresponding to identical boxes, and an IoU of 0 corresponding to boxes with no overlap. Fig. 10 shows an example of the bounding boxes returned by the base model and a reconstruction obtained using a parity model. For this example, the base model has an IoU of 0.88 and the reconstruction has an IoU of 0.61. This reconstruction captures the gist of the localization and would serve as a reasonable approximation in the face of unavailability. On the entire dataset, the base model achieves an average IoU of 0.95 with ground-truth bounding boxes. In degraded mode, using parity models with $k = 2$ achieves an average IoU of 0.67.

c) Varying parameter k: Fig. 11 shows that the degraded mode accuracy of parity models decreases as k increases from 2 to 4. As k increases, features from more queries are packed into a single parity query, making the parity query noisier and making it difficult to learn a parity model. This indicates a tradeoff between the value of parameter k and degraded mode accuracy.

To put these accuracies in perspective, consider a scenario in which no redundancy is used to mitigate unavailability. In this case, when the output from any base model is unavailable, the best one can do is to return a random "default" prediction. The option to provide such a default prediction is available in Clipper [39], an open-source prediction serving system. The degraded mode accuracy when returning default predictions depends on the number of possible outputs of an inference task. For example, a classification task with ten classes would have an expected degraded mode accuracy of 10% with this technique. Default predictions provide a lower bound on degraded mode accuracy and an indicator of the difficulty of a task. Fig. 11 shows that the degraded mode accuracy with parity models is significantly above this lower bound, indicating that parity models make significant progress in the task of reconstructing predictions.

F. Evaluation of tail latency reduction

We next briefly report on the the ability of parity models to reduce tail latency in machine learning inference. A more detailed evaluation of the latency reduction using parity models may be found in our prior work [15].

1) Evaluation setup: We have implemented parity models atop Clipper [39], a popular open-source prediction serving system. We call our system ParM. We implement the encoder and decoder on the Clipper frontend. Inference runs in Docker containers on backend "model instances," as is standard in Clipper. We use the addition/subtraction code described in Section V-A.

We consider as a baseline a system with the same number of instances as ParM but using all additional instances for deploying extra copies of the base model. We call this baseline "Equal-Resources." For a setting of parameter k on a cluster with m model instances for base models, both ParM and Equal-Resources use $\frac{m}{k}$ additional model instances. ParM uses these extra instances for parity models, whereas this baseline hosts extra base models on these instances. These extra instances enable the baseline to reduce load, which reduces tail latency.

Experiments are run on AWS EC2 on *p2.xlarge* nodes, which have one NVIDIA K80 GPU each. We use 12 instances for base models and $\frac{12}{k}$ additional instances for redundancy. We use one frontend of type *c5.9xlarge* and a single-queue load-balancing strategy for dispatching queries to model instances, as is standard in Clipper.

We evaluate latency using ResNet-18. We use ResNet-18 rather than a more computationally expensive model like ResNet-152 to provide a challenging scenario in which ParM must reconstruct predictions with low latency. Queries are images from the Cat v. Dog dataset. These higher-resolution images test the ability of ParM's encoder to operate with low latency. We modify base models and parity models to return 1000 values as predictions to create a computationally challenging decoding scenario in which there are 1000 classes.

We evaluate ParM's ability to mitigate slowdowns by inducing background load on the cluster running ParM. We emulate network traffic typical of data analytics

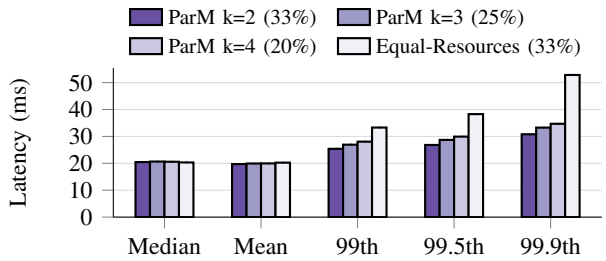


Fig. 12: Latencies of ParM at varying k compared to the strongest baseline. The portion of resources used for redundancy in each configuration is listed in parentheses.

workloads by choosing four pairs of model instances at random to transfer data between one another of size drawn randomly between 128-256 MB.

Clients send 100K queries to the frontend at a Poisson arrival rate with a mean of 270 queries per second. We measure the time between when the frontend receives a query and when the corresponding prediction is returned to the frontend (from a base model or reconstructed). We report the median of three runs. We use batch size of one, which is preferred for low-latency inference [40]. An evaluation of the latency of encoding and decoding in ParM may be found in our prior work [15].

2) Results: With $k = 2$, ParM reduces 99.9th percentile latency by up to 48% compared to Equal-Resources, bringing tail latency up to $3.5\times$ closer to median latency, while maintaining the same median.

Fig. 12 shows that ParM's tail latency increases as k increases. At higher values of k, ParM is more vulnerable to multiple predictions being unavailable, as the decoder requires k predictions to be available. Furthermore, increasing k increases the amount of time ParM needs to wait for k queries to arrive before encoding into a parity query. This increases the latency of the end-to-end path of reconstructing an unavailable prediction.

Despite these factors, ParM still reduces tail latency, even when using less resources than the baseline. With 25% and 20% redundancy, ParM reduces the gap between tail and median latency by up to $2.5\times$ compared to when Equal-Resources has 33% redundancy.

VI. Discussion

A. Differences from traditional, handcrafted codes

While this work shows the potential of taking a learning-based approach to coded computation, leveraging machine learning loses some of the benefits of traditional, handcrafted approaches to coded computation.

First, taking a learning-based approach results in the reconstruction of approximations of unavailable function outputs. In contrast, traditional coding techniques typically focus on exact recovery.

Second, our proposed learning-based approaches to coded computation require that learned components (i.e., encoder and decoder, or parity model) be retrained for every new computation $\mathcal{F}$ that is encountered. In contrast, traditional approaches to coded computation focus on the design of codes that are applicable to entire classes of computation (e.g., any linear computation).

We expect that further research may reduce the effects of these differences. For example, improvements in the training procedures presented in this work may reduce the accuracy loss incurred by learned components. Similarly, techniques like transfer learning may reduce the extent to which learned components must be retrained for each new computation $\mathcal{F}$.

B. Handling multiple unavailabilities

The evaluation in this work focuses on the scenario in which one out of $(k+r)$ servers is unavailable (i.e., $r = 1$). However, the proposed approaches can be extended to handle multiple unavailabilities (i.e., $r > 1$).

The first approach of learning encoders and decoders naturally accommodates multiple unavailabilities, as the setup is described for any value r in Section IV-A.

The second approach of learning parity models can tolerate multiple unavailabilities by training r separate parity models. For example, consider having $k = 2$, $r = 2$, and queries X_1 and X_2. One parity model can be trained to transform $P_1 = (X_1 + X_2)$ into $\mathcal{F}(X_1)+\mathcal{F}(X_2)$, while the second is trained to transform $P_2 = (X_1+2X_2)$ into $\mathcal{F}(X_1)+2\mathcal{F}(X_2)$. A decoder can reconstruct the initial k predictions using any k out of the $(k+r)$ predictions from base models and parity models.

C. Joint learning of encoder, decoder, and parity model

As depicted in Fig. 2 and Fig. 5, the two learning-based approaches proposed in this work learn disjoint sets of operations among the encoder, decoder, and computation over parities. In general, learning-based coded computation can be extended to jointly learn all of these components. Such joint optimization may help improve the accuracy of reconstructions while potentially reducing the computational overhead of the learned encoders and decoders described in Section IV-E.

VII. Conclusion

We proposed a *learning-based* approach to coded computation to overcome the challenge of applying coded computation to general non-linear functions. We present two paradigms in which learning can be used within the coded computation framework: (1) learning encoders and decoders, and (2) taking a fundamentally new approach to coded computation by learning a computation over parity units. We show that learning-based

coded computation enables accurate reconstruction of unavailable results from widely deployed neural networks for a variety of inference tasks such as image classification, speech recognition, and object localization. Our implementation and evaluation of learning-based coded computation on top of an open-source prediction serving system significantly reduces tail latency. These results highlight the potential of learning-based approaches to enable the benefits of coded computation for broader classes of non-linear computations.

VIII. Acknowledgments

This work was funded in part by an NSF Graduate Research Fellowship (DGE-1745016 and DGE-1252522), by NSF grants CNS-1850483 and CNS-1838733, by Amazon Web Services, and by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin, Madison with funding from the Wisconsin Alumni Research Foundation.

References

[1] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

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

[3] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding Up Distributed Machine Learning Using Codes," *IEEE Transactions on Information Theory*, July 2018.

[4] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *NIPS*, 2016.

[5] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A Unified Coding Framework for Distributed Computing With Straggling Servers," in *IEEE Globecom Workshops*, 2016.

[6] Q. Yu, M. Maddah-Ali, and S. Avestimehr, "Polynomial Codes: An Optimal Design for High-Dimensional Coded Matrix Multiplication," in *NIPS*, 2017.

[7] S. Wang, J. Liu, and N. Shroff, "Coded Sparse Matrix Multiplication," in *ICML*, 2018.

[8] S. Dutta, V. Cadambe, and P. Grover, "Coded Convolution for Parallel and Distributed Computing Within a Deadline," in *IEEE ISIT*, 2017.

[9] A. Reisizadeh, S. Prakash, R. Pedarsani, and S. Avestimehr, "Coded Computation Over Heterogeneous Clusters," in *IEEE ISIT*, 2017.

[10] A. Mallick, M. Chaudhari, G. Palanikumar, U. Sheth, and G. Joshi, "Rateless Codes for Near-Perfect Load Balancing in Distributed Matrix-vector Multiplication," in *ACM SIGMETRICS*, 2020.

[11] R. K. Maity, A. S. Rawat, and A. Mazumdar, "Robust Gradient Descent via Moment Encoding with LDPC Codes," in *IEEE ISIT*, 2019.

[12] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler Mitigation in Distributed Optimization Through Data Encoding," in *NIPS*, 2017.

[13] Q. Yu, N. Raviv, J. So, and A. S. Avestimehr, "Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy," in *AISTATS*, 2019.

[14] K. Konstantinidis and A. Ramamoorthy, "CAMR: Coded Aggregated MapReduce," in *IEEE ISIT*, 2019.

[15] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Parity Models: Erasure-Coded Resilience for Prediction Serving Systems," in *ACM SOSP*, 2019.

[16] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang, "LASER: A Scalable Response Prediction Platform for Online Advertising," in *WSDM*, 2014.

[17] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A Unified Coded Deep Neural Network Training Strategy Based on Generalized Polydot Codes for Matrix Multiplication," in *IEEE ISIT*, 2018.

[18] K. G. Narra, Z. Lin, G. Ananthanarayanan, S. Avestimehr, and M. Annavaram, "Collage Inference: Tolerating Stragglers in Distributed Neural Network Inference using Coding," *arXiv:1904.12222*, 2019.

[19] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Learning a Code: Machine Learning for Approximate Non-Linear Coded Computation," *arXiv preprint arXiv:1806.01259*, 2018.

[20] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *ICML*, 2017.

[21] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, "Improving Distributed Gradient Descent Using Reed-Solomon Codes," in *IEEE ISIT*, 2018.

[22] S. Li, S. M. M. Kalan, A. S. Avestimehr, and M. Soltanolkotabi, "Near-optimal Straggler Mitigation for Distributed Gradient Methods," in *IEEE IPDPSW*, 2018.

[23] E. Nachmani, E. Marciano, L. Lugosch, W. J. Gross, D. Burshtein, and Y. Be'ery, "Deep Learning Methods for Improved Decoding of Linear Codes," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 119–131, 2018.

[24] H. Kim, Y. Jiang, R. Rana, S. Kannan, S. Oh, and P. Viswanath, "Communication Algorithms via Deep Learning," in *ICLR*, 2018.

[25] F. A. Aoudia and J. Hoydis, "End-to-End Learning of Communications Systems Without a Channel Model," *arXiv:1804.02276*, 2018.

[26] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers," in *ACM SIGCOMM*, 2014.

[27] Fisher Yu and Vladlen Koltun, "Multi-Scale Context Aggregation by Dilated Convolutions," in *ICLR*, 2016.

[28] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms," *arXiv:1708.07747*, 2017.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE CVPR*, 2016.

[30] Z. Lin, R. Memisevic, and K. Konda, "How Far Can We Go Without Convolution: Improving Fully-Connected Networks," *arXiv:1511.02580*, 2015.

[31] Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015.

[32] X. Glorot and Y. Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," in *AISTATS*, 2010.

[33] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv:1611.01578*, 2016.

[34] J. Elson, J. R. Douceur, J. Howell, and J. Saul, "Asirra: A CAPTCHA That Exploits Interest-aligned Manual Image Categorization," in *ACM CCS*, 2007.

[35] P. Warden, "Speech commands: A Dataset for Limited-Vocabulary Speech Recognition," *arXiv:1804.03209*, 2018.

[36] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona, "Caltech-UCSD Birds 200," Tech. Rep. CNS-TR-2010-001, California Institute of Technology, 2010.

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[38] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.

[39] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System," in *USENIX NSDI*, 2017.

[40] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving RNN-based Deep Learning Models 10x Faster," in *USENIX ATC*, 2018.