Coded-InvNet for Resilient Prediction Serving Systems

Tuan Dinh [1] **Kangwook Lee** [2]

Abstract

Inspired by a new coded computation algorithm for invertible functions, we propose Coded-InvNet, a new approach to design resilient prediction serving systems that can gracefully handle stragglers or node failures. Coded-InvNet leverages recent findings in the deep learning literature such as invertible neural networks, Manifold Mixup, and domain translation algorithms, identifying interesting research directions that span across machine learning and systems. Our experimental results show that Coded-InvNet can outperform existing approaches, especially when the compute resource overhead is as low as 10%. For instance, without knowing which of ten workers is going to fail, our algorithm can design a backup task to correctly recover the missing prediction with an accuracy of 85.9%, significantly outperforming the previous SOTA by 32.5%.

1. Introduction

Prediction serving systems (PSSs) are systems that host pre-trained machine learning models for inference queries such as search, translation, and ranking ones. PSSs usually consist of a large number of parallel/distributed compute workers to process a large batch of input queries in parallel. As the system scales, maintaining low response time (latency) becomes more challenging. This is because the slowdown or failure of a single node can slow down the entire query processing time (Dean & Barroso, 2013; Narra et al., 2019). Multiple approaches for latency reduction have been proposed, including detect-and-relaunch (Zaharia et al., 2008), query replication (Suresh et al., 2015), and approximate inference (Han et al., 2017).

Coded computation, by introducing backup tasks in a coded form, improves system resilience with lower overhead compared to other methods (Lee et al., 2017). Recently, (Kosaian et al., 2019) showed that a coded computation-based technique, called Parity Models (ParMs), can significantly reduce the latency of PSSs. ParM consists of a tuple of encoder/parity model/decoder: the encoder aggregates multiple queries into a coded query, the parity model computes an inference task on it, and the decoder uses the encoded query together with available task results to reconstruct missing task results. However, experimental results of ParM were limited to small-scale PSSs (2–4 workers), and it is not clear whether the proposed method is applicable to large-scale PSSs. Also, small-scale regimes correspond to high resource overhead regimes, as one backup worker is required for every two to four workers. We empirically show that ParM does not scale well. For instance, when there is only one backup worker for ten main workers, ParM achieves approximately 19% accuracy on the CIFAR10 image classification when one of the main workers is not available.

We propose Coded-InvNet, a new coded computation-based framework to design scalable resilient prediction serving systems. Inspired by an efficient coded computation scheme for invertible functions, Coded-InvNet designs the inference function with a computationally-heavy but invertible module followed by a computationally-light module. To design the invertible module, we take advantage of recent developments in invertible neural networks (Behrmann et al., 2018; Song et al., 2019; Jacobsen et al., 2018). Then, by making use of GAN-based paired domain translation algorithms (Isola et al., 2017), Coded-InvNet trains a lightweight encoding function so that one can efficiently generate encoded queries from input queries without incurring high encoding overhead. To further improve the classification accuracy in the event of failures, Coded-InvNet also leverages mixup algorithms (Zhang et al., 2017; Verma et al., 2018).

We evaluate the efficacy of Coded-InvNet in image classification and multitask learning setting. Experimental results show that Coded-InvNet can scale much beyond the scales at which existing algorithms operate. More specifically, Coded-InvNet achieves higher reconstruction accuracy compared to ParM, and the accuracy gap increases remarkably as the system scales. For instance, Coded-InvNet can achieve reconstruction accuracy of 85.9% with 10 workers, while that of ParM is 53.4%. We also evaluate end-to-end laten-

[1]Department of Computer Sciences, University of Wisconsin-Madison, Madison, USA [2]Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, USA. Correspondence to: Tuan Dinh <tuan.dinh@wisc.edu>, Kangwook Lee <kangwook.lee@wisc.edu>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

cies on an AWS EC2 cluster, showing that Coded-InvNet's computing overhead is negligible.

1.1. Key Idea of Coded-InvNet

The key idea of coded computation is best illustrated when the target function is linear (Lee et al., 2017). For illustration purposes, consider two inputs x_1 and x_2 and three parallel workers. The goal here is to assign computation tasks to the workers so that one can obtain $f(x_1)$ and $f(x_2)$ even in the presence of slowdown or failure of a node. Coded computation assigns $f(x_1)$ to the first worker and $f(x_2)$ to the second worker. For the third worker, it first combines two input queries to get an encoded query $\frac{x_1+x_2}{2}$. It then assigns $f\left(\frac{x_1+x_2}{2}\right)$ to the third worker. By the linearity of the target function $f(\cdot)$, we have $f\left(\frac{x_1+x_2}{2}\right) = \frac{f(x_1)+f(x_2)}{2}$.

These three tasks can be viewed as three linearly independent weighted sums of $f(x_1)$ and $f(x_2)$, i.e.,

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ \frac{f(x_1)+f(x_2)}{2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f(x_1) \\ f(x_2) \end{bmatrix}. \tag{1}$$

Observe that *any* two rows of the coefficient matrix in the RHS of (1) is an invertible matrix. That is, as long as any two of the three computation results are available, the decoder $\mathrm{Dec}(\cdot,\cdot)$ can decode the computation results to recover $f(x_1)$ and $f(x_2)$. For instance, consider a situation where the second computation result $f(x_2)$ is missing. Then,

$$\begin{bmatrix} f(x_1) \\ \frac{f(x_1)+f(x_2)}{2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f(x_1) \\ f(x_2) \end{bmatrix}. \tag{2}$$

Since the coefficient matrix is full rank, the decoder can simply multiply the inverse of the coefficient matrix to recover $f(x_1)$ and $f(x_2)$.

Coded-InvNet is based on a simple observation that this coded computation framework is applicable to the family of invertible functions. Note that the target function is assumed to be linear or polynomial in the input in the existing works (Lee et al., 2017; Yu et al., 2019). Shown in Fig. 1 is the visual illustration of the coded computation algorithm for an invertible function. Consider the following encoding function $\mathrm{Enc}(x_1, x_2) = f^{-1}\left(\frac{f(x_1)+f(x_2)}{2}\right)$. By applying the target function to x_1, x_2, and $f^{-1}\left(\frac{f(x_1)+f(x_2)}{2}\right)$, we obtain $f(x_1)$, $f(x_2)$, and $\frac{f(x_1)+f(x_2)}{2}$. Therefore, as shown in the previous example, the decoder can always recover $f(x_1)$ and $f(x_2)$ with *any* two out of three task results.[1]

While this example demonstrates an efficient coded computation for invertible functions, this scheme 'as it is' cannot be used in practice. This is because the encoding process involves multiple evaluations of $f(\cdot)$ and $f^{-1}(\cdot)$, resulting in a significant encoding overhead.

[1] We provide a concrete example on synthesis dataset to complete this illustration in the supplementary material.

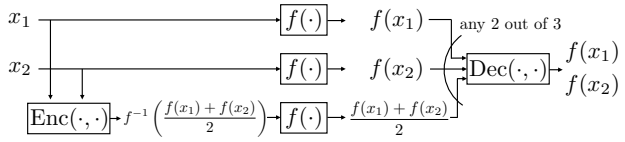

Figure 1. ***Coded computation for invertible functions.*** *Consider an invertible function $f(\cdot)$ and two inputs x_1 and x_2. The optimal encoding function $\mathrm{Enc}(\cdot,\cdot)$ takes the two inputs and generates an encoded input $f^{-1}\left(\frac{f(x_1)+f(x_2)}{2}\right)$. By applying function $f(\cdot)$ to the original inputs and the encoded input, one can obtain linearly independent weighted sum of $f(x_i)$'s. Thus, a simple decoder can always take* any *2 out of 3 computation results to decode $f(x_1)$ and $f(x_2)$ by solving a system of linear equations. For instance, if $f(x_2)$ is missing, one can multiply $\frac{f(x_1)+f(x_2)}{2}$ by 2 and then subtract $f(x_1)$ to recover $f(x_2)$. Our framework Coded-InvNet extends this idea to design resilient prediction serving systems.*

However, in machine learning applications, one can resolve this issue for the following reasons. First, $f(\cdot)$ does not need to be exactly computed, and it is sufficient to compute $f(\cdot)$ up to some approximation error. For instance, if $f(\cdot)$ is the feature extractor, then a small approximation error may not alter results of the following downstream tasks such as classification, segmentation, etc. This immediately implies that one can safely replace the computationally expensive encoding function with an approximate encoding function. Second, $f(\cdot)$ is also a design choice in machine learning applications. For instance, if $f(\cdot)$ is a feature extractor used by a downstream classification task, one can choose any $f(\cdot)$ as long as the classification performance is maintained.

Inspired by these observations, Coded-InvNet replaces the ideal encoding function with an approximate function that is parameterized by a light-weight neural network. Then, it *learns* the encoding function $\mathrm{Enc}(\cdot)$ together with the target function $f(\cdot)$. By carefully designing the encoding function's architecture, one can obtain low enough encoding approximation error while maintaining the encoding overhead negligible.

2. Related Works

Straggler mitigation in prediction serving systems has been studied in various approaches, such as detect-and-relaunch (Zaharia et al., 2008), replication (Dean & Barroso, 2013), approximate computing (Goiri et al., 2015), and coded computing (Lee et al., 2017). Among these, coded computation requires a minimal amount of compute resource overhead, making it promising with many methods to provide resiliency to slowdowns and failures in machine learning PSS (Lee et al., 2017; Li et al., 2016; Kosaian et al., 2018; 2019; Narra et al., 2019). Learning-

a-code (Kosaian et al., 2018; Dhakal et al., 2019) learns a neural network-based encoder/decoder pair to transform erasure-coded queries into a form that decoders can reconstruct unavailable predictions. However, this approach suffers from high encoding and decoding overhead. ParM (Kosaian et al., 2019) overcomes this limitation by learning a new inference function applied to encoded queries, which they call parity models. (Narra et al., 2019) proposes a novel convolutional neural network for multi-image classification on a collage image in one shot, thus reducing cost redundancy. This design is specific for image classification while Coded-InvNet is applicable to more downstream tasks.

Invertible Neural Networks (INNs) are NNs that can be inverted from the input up to the projection, or final classes (Jacobsen et al., 2018; Behrmann et al., 2018; Song et al., 2019). i-RevNets (Jacobsen et al., 2018) define an INN family based on a succession of homeomorphic layers. Behrmann et al. (2018) builds i-ResNet on top of residual networks (He et al., 2016), showing that one can invert a residual block with an exponential convergence rate via fixed-point iteration if Lipschitz constants of nonlinear parts are strictly less than 1. The authors bound each layer's Lipshitz constant by normalizing the weight matrix by its spectral norm. We adopt i-ResNet for our INN's architecture.

Manifold Mixup (Verma et al., 2018) extends Mixup (Zhang et al., 2017) to augment classification models with virtual data points which are random convex combinations of data points in the hidden space. This simple scheme effectively improves the margin in the hidden space, helps flatten the class-conditional representation, and reduces the number of directions by a significant variance. Coded-InvNet uses the classifier augmented with Manifold Mixup to make it more robust to encoding approximation errors.

Pix2Pix (Isola et al., 2017) is an image-to-image translation model that converts a set of images into a target image. Pix2Pix trains a conditional GAN (Mirza & Osindero, 2014) on pairs of input and target images, using a combination of GAN and L_1 losses. Pix2Pix uses Patch-GAN (Isola et al., 2017) and U-Net (Ronneberger et al., 2015) for the discriminator and generator architectures, respectively. We use Pix2Pix model for our encoder.

3. Coded-InvNet

3.1. Setting

Given a task needed to be deployed, we train an invertible network followed by a neural network that has significantly fewer layers and parameters compared to the first part. While this specific architectural choice may seem restrictive, it indeed supports a large range of applications. First of all, for classification tasks, state-of-the-art invertible NNs nearly match the performance of non-invertible NNs, and the gap is quickly closing. For instance, MintNet (Song et al., 2019) achieves only 1.4% lower accuracy on CIFAR10 than ResNet. Moreover, even non-invertible NNs are pseudo-invertible when trained with adversarial training (Engstrom et al., 2019), so our framework can serve any robust classification model. Though we limit our focus on the classification in this work, Coded-InvNet can be immediately extended to support generative models that are based on invertible architectures, such as flow-based models (Kingma & Dhariwal, 2018), or invertible Transformer-based models (Kim et al., 2020).

Shown in Fig. 2 is the PSS architecture of Coded-InvNet. Here, the overall inference network is denoted by $\hat{y}(x) := g(f(x))$, where $f(\cdot)$ is the invertible module of the neural network and $g(\cdot)$ is the second module of the network. The goal is to compute $\hat{y}_1 = g(f(x_1)), \hat{y}_2 = g(f(x_2)), \ldots, \hat{y}_k = g(f(x_k))$ in parallel with n ($n \geq k$) parallel workers in the presence of stragglers or failures. At the time the query arrives, the front end of the PSS applies $n-k$ encoding functions to obtain $n-k$ encoded queries. The i-th encoding function is denoted by $\mathrm{Enc}_i(\cdot)$, and the i-th encoded query is denoted by x_{k+i} for $1 \leq i \leq n-k$. That is, $x_1, x_2, \ldots, x_k$ denote the k inputs while $x_{k+1}, x_{k+2}, \ldots, x_n$ denote the $n-k$ encoded inputs.

Similar to the example illustrated in Sec. 1.1, we first identify the design of $n-k$ ideal encoding functions as follows. An ideal encoded input is such that one can get a linear combination of $f(x_i)$'s by applying $f(\cdot)$ to it. Let $f(x_{k+i}) = \sum_{j=1}^{k} c_{i,j} f(x_j)$ for $1 \leq i \leq n-k$. Then,

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_k) \\ f(x_{k+1}) \\ \vdots \\ f(x_n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n-k,1} & c_{n-k,2} & \cdots & c_{n-k,k} \end{bmatrix} \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_k) \end{bmatrix}. \quad (3)$$

To guarantee the decodability when any k tasks complete, any k rows of the coefficient matrix in the RHS of (3) must be full rank. For instance, when $n = k+1$, i.e., there is only one additional worker, one can satisfy this property by setting $c_{1,j} = \frac{1}{k}$ for all j. Also, when $c_{i,j}$ is chosen from the i.i.d. Gaussian distribution, one can show that the coefficient matrix satisfies the property with probability 1. Referring to the readers to (Bodmann, 2013) for various ways of choosing $c_{i,j}$'s, we will assume that $c_{i,j}$'s are chosen such that the desired property holds.

Remark 1. *When choosing $c_{i,j}$'s, in addition to the decodability condition, one should also consider whether $f(x_{k+i}) = \sum_{j=1}^{k} c_{i,j} f(x_j)$ lies close enough to the original manifolds in the embedding space. This is because the invertibility of f^{-1} may not hold for embedding vectors that are too far away from the original manifolds. Moreover, if*

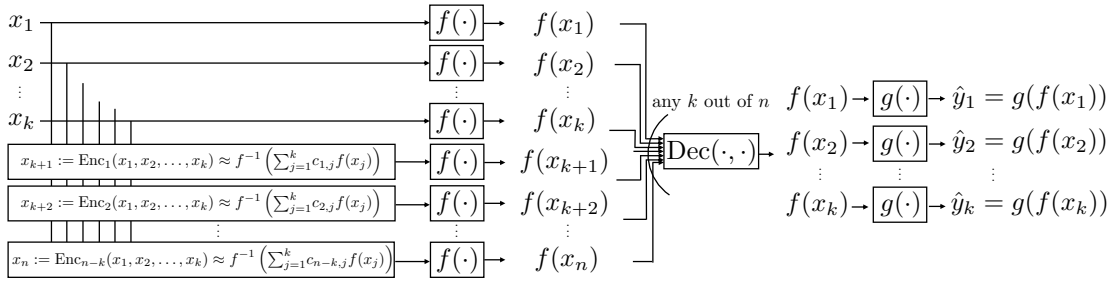

Figure 2. **Prediction serving system architecture of Coded-InvNet.** *Given k inputs $x_1, x_2, \ldots, x_k$, the goal of the predicting serving system is to compute $\hat{y}(x_i) := g(f(x_i))$ for all $1 \leq i \leq k$ in the presence of stragglers or failures, where $f(\cdot)$ is an invertible function. To this end, Coded-InvNet first generates $n-k$ encoded inputs $x_{k+1}, \ldots, x_n$ by applying $n-k$ distinct encoding functions to x_i's. It then assigns the task of computing $f(x_i)$ to each of n parallel workers. By leveraging the coded computation algorithm for invertible functions described in Sec. 1.1, one can approximately decode $f(x_1), f(x_2), \ldots, f(x_k)$ as soon as any k out of n tasks are completed. Note that the approximation errors occur since we use approximate encoding functions. Once $f(x_1), f(x_2), \ldots, f(x_k)$ are computed, the front end applies $g(\cdot)$ to each of them and returns the query results.*

this additional condition holds, $f^{-1}(f(x_{k+i}))$ might share some semantic features with the original data. This essentially reduces the gap between the domain where original inputs belong and the domain where encoded inputs belong, allowing for efficient learning of encoding functions with domain translation techniques.

3.2. Approximate Encoding Functions

Recall that ideal encoding functions cannot be used in practice due to their large computation overhead. Thus, we use a neural network (NN) to approximate ideal encoding functions. The universal approximation theorem asserts that ideal encoding functions can be well approximated by some NNs (Cybenko, 1989). By limiting the number of layers and parameters of the NN, we can control the computation overhead of the NN-based approximate encoding functions.

More specifically, for each i, $1 \leq i \leq n-k$, we want to train a neural network $\mathrm{Enc}_i(\cdot)$ such that $\mathrm{Enc}_i(\cdot) \approx f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)$. Note that if $x \in \mathcal{X}$, then $\mathrm{Enc}_i(\cdot) : \mathcal{X}^k \to \mathcal{X}$, i.e., the encoding function takes k inputs from domain $\mathcal{X}$ to generate one output in $\mathcal{X}$. Thus, to learn the i-th encoding function, we need to collect the following train set: $\left\{(x_1, x_2, \ldots, x_k), f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)\right\}$. Here, the input $(x_1, x_2, \ldots, x_k)$ is a tuple of k randomly chosen inputs from the train set. Given the input tuple, the 'label' $f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)$ can be always computed by applying $f(\cdot)$ and $f^{-1}(\cdot)$. Once the train set for an encoding function is collected, one can train the encoding function such that $\mathrm{Enc}_i(\cdot) \approx f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)$ holds on the collected train set.

One can also make use of explicit semantic loss when training encoding functions. For illustration purposes, assume that $n = k+1$ and $c_{1,j} = \frac{1}{k}$ for all j. When $f(x_a)$ is missing for some $1 \leq a \leq k$, and all the other task results are available, one can decode $f(x_i)$ as follows: $\widehat{f(x_a)} = kf(x_{k+1}) - \sum_{i=1,\ldots,k,i\neq a}^{k} f(x_i)$. And the inference result will be $g\left(\widehat{f(x_a)}\right)$. By comparing this inference result with the correct result $g(f(x_a))$, one can explicitly capture the semantic difference between them. For instance, if the target task is classification, one can compare the logit values of these outputs and apply the distillation loss (Hinton et al., 2015). Note that one can also 'co-train' these encoding functions together with the target function $g \circ f$.

Remark 2. *One can further reduce the inference overhead of the encoder by applying neural network compression techniques such as pruning, knowledge distillation, vector quantization, etc. Such techniques allow for larger encoder architectures, reducing the encoding error.*

Remark 3. *One can deploy* $\mathrm{Enc}(\cdot,\cdot)$ *at the front-end server or at the backup workers. The former option increases the computational load of the front-end server while the latter increases the load of backup workers. Moreover, the latter incurs extra communication cost between the front-end and the workers. Thus, one should make a proper choice considering the communication/computation tradeoff.*

3.3. Minimizing Encoding Error Propagation

Once we introduce approximate encoding functions into this framework, $f(x_{k+i})$, for $1 \leq i \leq n-k$, will not exactly match the desired linear combination $\sum_{j=1}^{k} c_{i,j} f(x_j)$. That is, with approximate encoding, we only have $x_{k+i} \approx f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)$, so $f(x_{k+i}) = \sum_{j=1}^{k} c_{i,j} f(x_j) + \varepsilon$, where ε is the approximation error in f, which depends on $(x_1, x_2, \ldots, x_k)$.

To see how this approximation error affects the inference quality, assume that $n = k+1$ and consider a scenario where $f(x_1)$ is missing, and $f(x_2), f(x_3), \ldots, f(x_k), f(x_{k+1})$

are available. By assuming $c_{1,j} = \frac{1}{k}$, one can decode $f(x_1)$ as follows: $\widehat{f(x_1)} = kf(x_{k+1}) - \sum_{i=2}^{k} f(x_i)$. Since $f(x_{k+1}) = \sum_{j=1}^{k} c_{1,j} f(x_j) + \varepsilon$, we have $\widehat{f(x_1)} = f(x_{k+1}) + k\varepsilon$. Then, $\hat{y}_1 = g(f(x_{k+1}) + k\varepsilon)$. Therefore, the recovered inference result for a missing computation task will be incorrect if $\hat{y}_1 = g(f(x_{k+1}) + k\varepsilon) \neq g(f(x_{k+1}))$.

To prevent this, one must first make sure that the encoding functions are well trained such that $\mathrm{Enc}_i(\cdot) \approx f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)$ holds. That way, one can expect that the magnitude of ε is small enough.

However, even when encoding error ε is small, if the second part of the inference module $g(\cdot)$ is highly sensitive to small perturbations, inference results could still be incorrect. Indeed, Verma et al. (2018) shows that standard deep neural networks are highly sensitive to small noises injected in the embedding space. This implies that a small encoding error, which necessarily arises during the approximate encoding procedure, can distort the overall inference result.

To resolve this issue, we leverage the recent Manifold Mixup (Verma et al., 2018) regularization technique, which smoothens class-conditional embedding spaces, making inference outputs more robust to noises injected in the embedding space. Inspired by this, Coded-InvNet trains $f(\cdot)$ and $g(\cdot)$ with Manifold Mixup to ensure that $\hat{y}_1 = g(f(x_{k+1}) + k\varepsilon) = g(f(x_{k+1}))$ with high probability.

3.4. Advantages of Coded-InvNet

Separation between Encoding and Inference By design, Coded-InvNet maintains a clear separation between encoding and inference,[2] which is highly advantageous when building a prediction serving system. First, every worker does the same work, i.e., simply computing $f(\cdot)$, simplifying the system implementation and management. On the other hand, ParM requires a highly heterogeneous system configuration: the first k workers compute $f(\cdot)$, and the other $n-k$ workers compute $n-k$ *distinct* parity models, say $f'_1(\cdot), \ldots, f'_{n-k}(\cdot)$. Similarly, when the total number of workers n and the total number of inputs k change, Coded-InvNet does not require any change to the worker configuration and simply needs to retrain the encoder.

Second, when optimizing inference time via various techniques such as model compression (Zhang et al., 2019), hardware optimization (Marculescu et al., 2018), or network pruning (Blalock et al., 2020), one just needs to focus on optimizing one inference function $f(\cdot)$. On the other hand, ParM needs to optimize $n-k+1$ models, incurring a significantly larger cost.

[2] Learning-a-code (Kosaian et al., 2018) also has this property.

Applicability to Multi-task Serving A representation in NNs can serve for various downstream tasks (Liu et al., 2015). For instance, well-trained image representation can be simultaneously useful for image classification, segmentation, and depth estimation (Kendall et al., 2018). For a detailed overview of multi-task learning and the role of shared representation, we refer the readers to (Ruder, 2017).

We highlight that Coded-InvNet is a promising solution when the prediction serving system is serving multiple inference tasks sharing the underlying representation. Recall that Coded-InvNet reconstructs the embedding (or representation) of missing inputs while existing approaches directly reconstruct missing inference results (Kosaian et al., 2018; 2019). Therefore, Coded-InvNet does not need any extra efforts to support multiple tasks, say m tasks. The only difference is that we now have $g_i(\cdot)$ for $1 \leq i \leq m$, where $g_i(\cdot)$ is specific to task i. Note that this does not affect the training complexity of the encoder at all. On the other hand, ParM has to train m parity models to achieve the same goal. See Sec. 4.6 for experimental results where we demonstrate the applicability of Coded-InvNet to multi-task settings.

4. Experiments

We show how we implement the components of the Coded-InvNet framework and evaluate its performance. In comparison with the baselines, we focus on the image classification task on popular datasets: MNIST (Deng, 2012), Fashion-MNIST (Xiao et al., 2017), and CIFAR10 (Krizhevsky & Hinton, 2009). These are all 10-way image classification tasks. We also demonstrate the applicability of Coded-InvNet on a large-scale ImageNet-based dataset (Howard, 2019), and on the multiple-failure setting.

We report various performance metrics such as the classification accuracy when straggler/failure happens, encoding/decoding overhead, scalability, end-to-end latency, etc.

Since Coded-InvNet recovers the full embedding of the missing input, it naturally fits with multi-task applications, i.e., one common embedding can be used for multiple downstream tasks. We also show how one can apply Coded-InvNet for such multi-task applications.

We provide further additional experiment results and implementation details in the supplementary material.

4.1. Architectures and Training Methods

While the Coded-InvNet framework is applicable to any values of n, k, $n \geq k$, the most important case is when $n = k+1$. This is of practical interest since this scheme's compute overhead is minimal ($\frac{100}{k}\%$) while still being robust against a single failure. We will consider $k \in \{2, 4, 10\}$ and choose $c_{1,j} = \frac{1}{k}$ for all j.

Architecture. We use i-ResNet as the classification network $g \circ f$.[3] We mostly follow the recommended configurations in (Behrmann et al., 2018), but we remove the injective padding module to improve the invertibility of off-manifold embedding vectors. We use the Pix2Pix (Isola et al., 2017) architecture for encoding functions in our experiments for all values of k. To avoid linear scaling, we design our encoder architecture such that only the complexity of the first few layers depends on k, and the rest of the architecture does not scale with k. See Fig. 3 for the encoder architecture for $k = 2$. When $k = 2$, two inputs x_1 and x_2 are first processed by the weight-shared network. The two processing results are then concatenated and projected to a fixed-size hidden vector. In general, our encoder processes k inputs in parallel, and the concatenated output is projected to the same size hidden vector. By limiting the size of this input processing part, we control how the encoder complexity scales as k increases. See Sec. 4.4 for experimental results where we demonstrate that encoding overhead can be kept nearly constant for increasing values of k.

Training. We train our i-ResNet classifier with Manifold Mixup (with mixup coefficient 1). Then, we use the trained classifier to generate a train set for the encoding network. In particular, we draw k random inputs $(x_1, x_2, \ldots, x_k)$ and compute labels $f^{-1}\left(\sum_{j=1}^{k} c_{i,j} f(x_j)\right)$ to obtain an input/output pair. Here, since our $f(\cdot)$ is an i-ResNet module, the inverse function does not have an explicit form, we compute the inverse function by solving fixed point equations. We repeat this $50,000$ times to construct a sufficiently large training set. We, then, train the encoder function using Least Square GAN loss (Mao et al., 2017) plus 100 times L_1 loss. We enforce permutation invariance of inputs by (1) sharing weights of the first hidden layers across all k inputs and (2) using the average function after the first hidden layer. The encoder training procedure is summarized in Fig. 3. Training the encoder may take up to 7.5 hours (ResNet-301-based architecture with 150 epochs on a 48-GB RTX8000 GPU).

4.2. Training Results

Classifier The trained classifiers achieve 99.2%, 91.8%, and 90.9% on MNIST, Fashion-MNIST, and CIFAR10, respectively. Note that removing the injection padding layer from the original i-ResNet, as mentioned previously, results in slight drops in the classification accuracy.

Encoder To see if the encoder training was successful, we observe the training curve (shown in Fig. 3 in the supplementary material). Here, we show the L_1 loss measured on the train/test sets of MNIST, respectively. The encoder was trained with $k = 4$ and $c_{1,j} = \frac{1}{4}$. The train L_1 loss keeps

[3] We report additional results with another invertible network i-RevNet (Jacobsen et al., 2018) in the supplementary material.

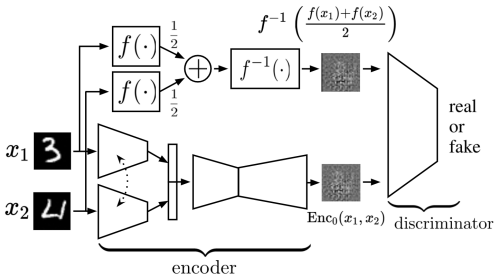

Figure 3. ***Encoder training.*** *Let $k = 2$ and $c_{1,1} = c_{1,2} = \frac{1}{2}$. We first pick two random inputs x_1 and x_2. We first apply $f(\cdot)$ to each of them, compute the average, and then apply $f^{-1}(\cdot)$, obtaining the target. We then feed the two inputs to the encoder. We ensure the input symmetry of the encoder by individually processing them with a weight-shared network. The processing results are concatenated and then get further processed to obtain the encoded input. We then train the encoder's weights with the Pix2Pix loss.*

improving while the test L_1 loss saturates around epoch 20. Shown in Fig. 4 is the qualitative performance evaluation of the trained encoder. In Fig. 4 (h), we show eight random samples of (x_1, x_2, x_3, x_4) from the test set. Fig. 4 (i) visualizes ideal encoded inputs. Recall that each of these ideal encoded inputs is computed by applying $f(\cdot)$ to each of x_i's, obtaining a weighted sum, and then computing $f^{-1}(\cdot)$. Since i-ResNet does not have an explicit inverse function, $f^{-1}(\cdot)$ is computed by solving a fixed point equation via an interactive method. In Fig. 4 (j), we show that the encoded inputs obtained by our trained encoder are indistinguishable from ideal encoding results to naked eyes.

To further check the validity of the trained encoder, we perform a qualitative evaluation by observing the visual quality of input decoding. Note that with the ideal encoding input, one can perfectly recover missing inputs. Assuming $n = k + 1$ and $c_{1,j} = \frac{1}{k}$, let us define

$$\widehat{f(x_1)} := kf\left(\mathrm{Enc}_0(x_1, x_2, \ldots, x_k)\right) - \textstyle\sum_{i=2}^{k} f(x_k), \tag{4}$$

$$\widehat{x_1} := f^{-1}\left(kf\left(\mathrm{Enc}_0(x_1, x_2, \ldots, x_k)\right) - \textstyle\sum_{i=2}^{k} f(x_k)\right). \tag{5}$$

That is, $\widehat{f(x_1)}$ is the reconstruction of the missing function evaluation, and $\widehat{x_1}$ is the reconstruction of the corresponding input. One can define $\widehat{f(x_i)}$ and $\widehat{x_i}$ for $2 \leq i \leq k$ in a similar way. If the encoding function is ideal, i.e., $\mathrm{Enc}_0\left(x_1, x_2, \ldots, x_k\right) = f^{-1}\left(\sum_{j=1}^{k} f(x_j)/k\right)$, then $\widehat{f(x_i)} = f(x_i)$ and hence $\widehat{x_i} = x_i$. Therefore, we can indirectly evaluate the performance of an encoder by comparing $\widehat{x_i}$ and x_i. See Fig. 4 (c), (d) for the comparison results for $k = 2$ and Fig. 4 (i), (j) for $k = 4$. Observe that $\widehat{x_i}$ closely recovers x_i in all tested cases up to some offsets, justifying the validity of the trained encoder.

4.3. Degraded Mode Accuracy

Comparison with Baselines We now evaluate the classification performance of Coded-InvNet under the presence of

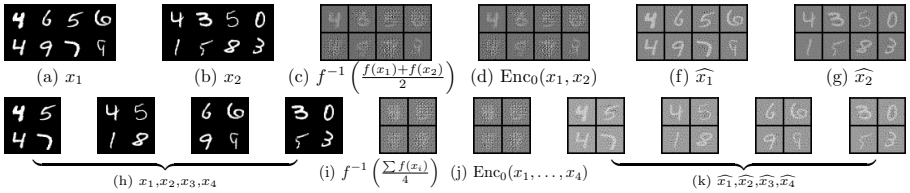

Figure 4. **Qualitative performance evaluation of a trained encoder for MNIST ($k = 2$ and $k = 4$).** *We used $c_{1,j} = \frac{1}{k}$. (a), (b): Randomly chosen x_1 and x_2. (c): The ideal encoding results, i.e., $f^{-1}((f(x_1) + f(x_2))/2)$. (d): The output of a trained encoder, i.e., $\text{Enc}_0(x_1, x_2)$. Note that (c) and (d) are visually indistinguishable. (f), (g): Note that $x_1 := f^{-1}\left(2f\left(f^{-1}((f(x_1) + f(x_2))/2)\right) - f(x_2)\right)$. By replacing $f^{-1}((f(x_1) + f(x_2))/2)$ with $\text{Enc}_0(x_1, x_2)$, we have $\widehat{x_1} := f^{-1}(2f(\text{Enc}_0(x_1, x_2)) - f(x_2))$. Similarly, one can also define $\widehat{x_2}$. Shown in (f) and (g) are $\widehat{x_1}$ and $\widehat{x_2}$. Observe that they successfully recover the missing inputs up to offsets. (h)–(k): We repeated the same experiment for $k = 4$ and observed similar results.*

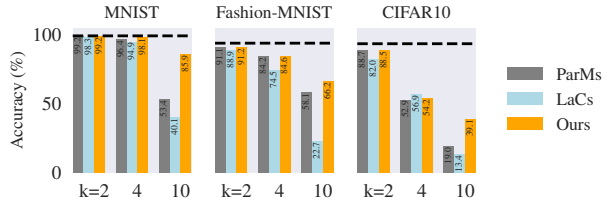

Figure 5. **Degraded mode accuracy comparison.** *We compare ParM, Learning-a-code (LaC), and Coded-InvNet (Ours) on different resource overhead $k = 2, 4, 10$. Dotted lines represent our normal accuracy $\mathbb{E}_{x,y}[\hat{y} = y]$, i.e., the classification accuracy when the inference does not involve any reconstruction. Bars represent the degraded mode accuracy $\mathbb{E}_{x_1,y_1,x_2,x_3,\ldots,x_k}[\hat{y} = y]$. Observe that all models achieve comparable performances when the resource overhead is high, e.g., $k = 2$ (50%) and $k = 4$ (25%). However, when the resource overhead is very low, i.e., $k = 10$ (10%), Coded-InvNet significantly outperforms on every dataset.*

stragglers or failures. *Normal accuracy* is simply the top-1 classification accuracy of our classifier, i.e., $\mathbb{E}_{x,y}[\hat{y} = y]$. If $f(x_i)$ is available at the time of k tasks complete, then $\widehat{f(x_i)} = f(x_i)$. Therefore, the normal accuracy will capture the classification accuracy for such cases.

We also define *degraded mode accuracy* (Kosaian et al., 2019). If any of $f(x_i)$ is not available at the time of k task results are available, we first decode $\widehat{f(x_i)}$ using (4) and then compute $\widehat{y_i} = g(\widehat{f(x_i)})$, the recovered inference result on x_i. Due to the encoding error, $\widehat{y_i}$ could be different from $g(f(x_1))$. The degraded accuracy is the accuracy measured with respect to this recovered inference result $g(\widehat{f(x_1)})$. We expect that the degraded accuracy is lower than the normal accuracy. More formally, it is defined as $\mathbb{E}_{x_1,y_1,x_2,x_3,\ldots,x_k}\left[g(\widehat{f(x_1)}) = y_1\right]$, where the expectation is over k random inputs.

We report the normal and degraded mode accuracy in Fig. 5. We compare our model with ParM (Kosaian et al., 2019) built with the default summation encoder/subtraction decoder,[4] and Learning-a-code (Kosaian et al., 2018) built with convolutional encoder/fully-connected decoder. The dotted lines represent normal accuracies on each dataset, and the bars are for the degraded mode accuracies.

When $k = 2$ or $k = 4$, all approaches obtain similar degraded accuracies on every dataset. When $k = 10$, i.e., the system is larger and the resource overhead is much lower ($1/k = 10\%$), a large number of embedding vectors have to be packed into a single aggregated vector. Also, the encoded input has more chance to be off the input manifold so is the corresponding embedding. These factors make the encoder learning more challenging, indicating a trade-off between resource overhead and degraded accuracy, i.e., as the resource overhead decreases (as k increases), the encoding and the degraded mode errors increase. Indeed, in the original paper, the state-of-the-art approach ParM was tested only for $k \leq 4$, and it was not clear how well this approach would perform in such a low resource regime.

As shown in Fig. 5, Coded-InvNet significantly outperforms ParM and Learning-a-code in the low resource regime. For instance, on MNIST, Coded-InvNet achieves 85.9% degraded accuracy while ParM and Learning-a-code achieve $53.4\%, 40.1\%$ respectively. On CIFAR10, the degraded mode accuracy of Coded-InvNet is 39.1%, more than twice that of ParM, and three times that of Learning-a-code.

Take-away: Coded-InvNet is highly resilient in the presence of stragglers or failures. It matches the performance of state of the art when the resource overhead is high enough ($\geq 25\%$), and significantly outperforms them when the resource overhead is as low as 10%.

Coded-InvNet on Large-scale ImageNet Data We show that our framework can be designed for more complex datasets. In particular, we use Imagenette that is a 10-class subset of ImageNet. We replace i-ResNet with an i-RevNet (Jacobsen et al., 2018) (96.2% in normal accuracy). We achieve 85.5% degraded accuracy for $(n, k) = (3, 2)$ and 58.4% for $(n, k) = (5, 4)$. Fig. 6 shows the sample reconstruction of our algorithm when $f(x_1)$ needs to be

[4] Kosaian et al. (2019) also develops a downsample-and-concatenation-based encoder for image classification, but this design is task-specific, and currently does not support $k = 10$.

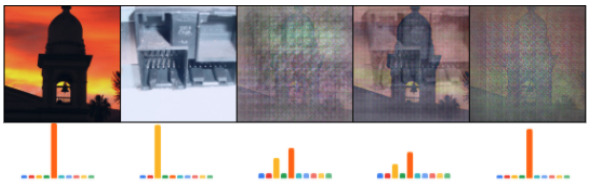

Figure 6. **Reconstruction on Imagenette samples**. *Left to right:* x_1, x_2, $f^{-1}((f(x_1)+f(x_2))/2)$, *Enc*$(x_1, x_2)$, *and* $\widehat{x_1}$. *Shown below them are the softmax scores of their prediction outputs* $f(\cdot)$. *Note that* $\arg\max f(\widehat{x_1}) = \arg\max f(x_1)$, i.e., *the degraded classification is correct.*

recovered from the backup worker. One can see that the recovered computation result (5th column) closely matches the missing computation result $f(x_1)$ (1st column).

Coded-InvNet with Multiple Failures We first handle two failures at the same time by applying Coded-InvNet with $(n, k) = (4, 2)$. We consider the following encoders:

$$\text{Enc}_1(x_1, x_2) = f^{-1}\left(\frac{f(x_1)+f(x_2)}{2}\right),$$
$$\text{Enc}_2(x_1, x_2) = f^{-1}\left(\frac{f(x_1)+2f(x_2)}{3}\right).$$

With these encoders, the four tasks can be viewed as four linearly independent sums of $f(x_1)$ and $f(x_2)$, *i.e.*,

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ \frac{f(x_1)+f(x_2)}{2} \\ \frac{f(x_1)+2f(x_2)}{3} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1/2 & 1/2 \\ 1/3 & 2/3 \end{bmatrix} \begin{bmatrix} f(x_1) \\ f(x_2) \end{bmatrix}.$$

One can confirm that *any* two rows of the coefficient matrix in the RHS is an invertible matrix. That is, as long as any two of the four computation results are available, the decoder can recover the computation results $f(x_1)$ and $f(x_2)$. We evaluate this coding scheme on the MNIST dataset. We consider a setting where both $f(x_1)$ and $f(x_2)$ are unavailable, *i.e.*, one must recover them only from the backup task results. Even for this very non-trivial setting, we obtain 67.4% degraded accuracy. Similarly, we also test our framework with $(n, k) = (6, 4)$. Focusing on the setting where two of the first four results are not available (i.e., both the encoders are involved when decoding), we achieve 45.4% degraded accuracy. To the best of our knowledge, all the previous works focused on the $n = k + 1$ setting, and this is the first non-trivial degraded accuracy for $n > k + 1$.

4.4. Encoding/decoding Overhead

At the time of query arrival, Coded-InvNet must first compute $n - k$ encoded queries before assigning compute tasks to workers. If the encoding procedure requires a non-negligible amount of compute time, then the whole premise of coded computation is invalidated.

To avoid this, as described in Sec. 4.1, we carefully chose the encoder architecture so that its compute time is negligible compared to that of the inference function. Shown in Fig. 7 are the wall-clock compute time of $\text{Enc}(\cdot)$, $f(\cdot)$ and $g(\cdot)$. We measure the running time over 10 runs and report the mean and standard deviation. All of these are measured on a 12-GB NVIDIA TITAN Xp GPU, 128-GB of DRAM, and 40 Intel Xeon E5-2660 CPUs. Note that the time to compute $f(\cdot)$ is dominant, and the encoding time is relatively negligible. The encoding overhead is 5.8% for MNIST, 8.3% for Fashion-MNIST, and 6.6% for CIFAR10.

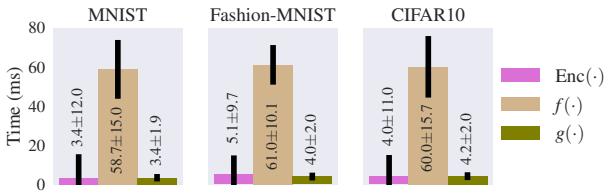

Figure 7. **Wall-clock compute time of** $\text{Enc}(\cdot)$, $f(\cdot)$, **and** $g(\cdot)$. *The compute overhead of encoding must be negligible compared to the actual inference time. We measure the wall-clock time to run* $\text{Enc}(\cdot)$, $f(\cdot)$, *and* $g(\cdot)$ *to quantify the compute overhead of encoding when* $k = 2$. *We fix the batch size as* 256 *images and measure the compute time over* 10 *independent runs, reporting averages, and standard deviations. We can observe low overheads:* 5.8% *for MNIST,* 8.3% *for Fashion-MNIST, and* 6.6% *for CIFAR10.*

Figure 8. **Encoding overhead as a function of** k. *The encoding overhead is measured for* $k = 2, 4, 10$. *We fix the batch size as* 256 *images and measure the compute time of* $\text{Enc}(\cdot)$ *over* 10 *independent runs, reporting the medians. The encoding overhead remains nearly constant as* k *increases on all datasets.*

As k increases, the encoder complexity also increases. To avoid linearly increasing complexity, in Sec. 4.1, we designed our encoder architecture such that only the first few layers' complexity depends on k. Shown in Fig. 8 is the encoding overhead as a function of k. Observe that the encoding overhead remains almost constant as k increases.

The decoding overhead of Coded-InvNet is minimal, and a simple online decoding algorithm can make it independent of k. See the supplemental materials for more details.

Take-away: Coded-InvNet's encoding overhead is negligible ($< 10\%$*) due to its lightweight encoder architecture and remains almost constant as* k *increases. The decoding overhead does not scale with* k *with online decoding.*

4.5. End-to-End Latency

We evaluate the effectiveness of Coded-InvNet in reducing tail latency by evaluating its end-to-end inference latency measured on an AWS EC2 cluster. In particular, we implement a custom testbed for Coded-InvNet and ParM, written with MPI4py (Dalcin et al., 2011). All experiments are

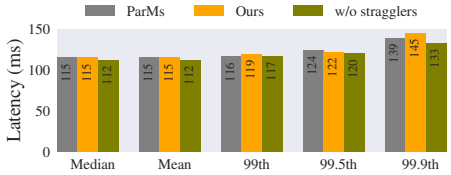

Figure 9. **End-to-end latency on AWS cluster with** $(n, k) = (11, 10)$. *Latencies are measured on ParM (grey), our Coded-InvNet (orange) and inference models without stragglers (olive). Coded-InvNet shows negligible overhead compared to ParM and inference models. For instance, the* 99th*-tail latency of Coded-InvNet is only* 3ms *higher than ParM, accounting for less than* 3% *inference latency. Note that, our extra latencies (and ParMs) probably come from the communication to the redundancy worker.*

run on an Amazon AWS EC2 cluster with GPU instances (p2.xlarge with NVIDIA K80). We deploy a pretrained inference model in $k = 10$ instances and deploy the encoder and the inference model in one additional instance. We also deploy one dedicated instance as the front-end node. To emulate stragglers in our testbed, we randomly add artificial latencies of 0.1s to one of the k workers, following the setup proposed in (Tandon et al., 2017). We then measure the end-to-end latency that is the time between when the front-end server receives a query and when the prediction result becomes available at the front-end server. We measure latencies over 5,000 independent CIFAR10 classification queries. ParM is set to use the default linear encoder/decoder and the same neural network architecture with ours. Shown in Fig. 9 are the end-to-end latency statistics of Coded-InvNet and ParM, compared with the latency statistics without artificial delays. As expected, Coded-InvNet's additional overhead is almost negligible, compared to the other latency sources such as communication and inference.

Varying Batch Size For completeness, we evaluate end-to-end latency of Coded-InvNet with batched queries. Using $(n, k) = (3, 2)$, with the batch size of 2 and 4, we also observe close gaps of 2.29% and 3.78% between Coded-InvNet and ParM on 99.9th percentile latency, respectively.

4.6. Applicability to Multi-task Serving

To illustrate the multi-tasking advantage, we perform two 2-task experiments on CIFAR10: (1) fine and coarse classifications, (2) classification, and density estimation. The overall architecture remains the same except that we have $g_1(\cdot), g_2(\cdot)$ for two tasks instead of only one $g(\cdot)$. In the first experiment, fine classification is with the 10 original classes, and we define two coarse classes as 'vehicle' (airplane, automobile, ship, and truck) and 'animal' (the other classes). Coded-InvNet achieves high accuracies for both the tasks, across different k (shown in Table 1 in the supplementary material). For instance, with $k = 4$, we achieve $86.6\%, 43.4\%$ on coarse and fine classification, respectively. For the second experiment, we co-train a classifier and a normalizing flow model (Behrmann et al., 2018) based on i-ResNet, obtaining 85.2% normal accuracy and 3.9 bits/dim, respectively. Applying Coded-InvNet, with $(n, k) = (5, 4)$, the two models recover 50% degraded accuracy and 6.8 degraded bits/dim. These elementary results demonstrate the applicability of Coded-InvNet to multi-task settings.

4.7. Ablation Study

Beyond Invertibility Extending our framework to non-invertible architectures is a crucial future direction. (Engstrom et al., 2019) shows that a robust representation learned by adversarial training is approximately invertible. When $f()$ is not invertible, one can still apply our framework by finding an approximate inverse of y by solving $\min_x \|f(x) - y\|$. We replace the INN in Coded-InvNet by the robust classifier (Engstrom et al., 2019) and use approximate inverse for inversion. On CIFAR10 with $(n, k) = (5, 4)$, we achieve 38.5% in degraded accuracy, which shows the promise of this direction.

Effect of Manifold Mixup The regularizer makes $g(\cdot)$ robust to small approximation errors. We found that degraded accuracies on CIFAR10 when $k = 2, 4$ decrease by 42% and 64% when trained without Manifold Mixup, despite gaining higher normal accuracy. As discussed in Sec. 3.3, this can be accounted to the effect of Manifold Mixup on learning smooth data manifold. This effect increases the chance that the aggregated embedding locates on the manifold, and thus the corresponding inverses obtain the in-domain structural and semantic representation. This makes encoder training easier as most domain translation algorithms are designed for close-to-enough domains.

5. Conclusion

We present Coded-InvNet, a new coded computation-based approach for designing a resilient prediction serving system. Inspired by a new coded computation algorithm for invertible functions, it jointly trains the inference function and the encoder, by making use of recent developments in the deep learning literature such as Manifold Mixup and domain translation algorithms. Maintaining a clear separation between encoding and inference, Coded-InvNet provides a hassle-free design alternative to the existing methods for system designers. Coded-InvNet is shown to significantly outperform the state of the art especially when the compute resource overhead is as low as 10%.

Acknowledgements

This material is based upon work supported by NSF/Intel Partnership on Machine Learning for Wireless Networking Program under Grant No. CNS-2003129.

References

Behrmann, J., Grathwohl, W., Chen, R. T., Duvenaud, D., and Jacobsen, J.-H. Invertible residual networks. *arXiv preprint arXiv:1811.00995*, 2018.

Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

Bodmann, B. G. *Frames as Codes*, pp. 241–266. Birkhäuser Boston, Boston, 2013. ISBN 978-0-8176-8373-3. doi: 10.1007/978-0-8176-8373-3_7. URL https://doi.org/10.1007/978-0-8176-8373-3_7.

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Dalcin, L. D., Paz, R. R., Kler, P. A., and Cosimo, A. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

Dean, J. and Barroso, L. A. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Dhakal, S., Prakash, S., Yona, Y., Talwar, S., and Himayat, N. Coded federated learning. *2019 IEEE Globecom Workshops (GC Wkshps)*, Dec 2019. doi: 10.1109/gcwkshps45667.2019.9024521. URL http://dx.doi.org/10.1109/GCWkshps45667.2019.9024521.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Tran, B., and Madry, A. Adversarial robustness as a prior for learned representations, 2019.

Goiri, I., Bianchini, R., Nagarakatte, S., and Nguyen, T. D. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 383–397, 2015.

Han, R., Huang, S., Wang, Z., and Zhan, J. Clap: Component-level approximate processing for low tail latency and high result accuracy in cloud online services. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2190–2203, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Howard, J. Imagenette dataset. 2019. URL https://github.com/fastai/imagenette.

Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1125–1134, 2017.

Jacobsen, J.-H., Smeulders, A., and Oyallon, E. i-revnet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

Kendall, A., Gal, Y., and Cipolla, R. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7482–7491, 2018.

Kim, H., Papamakarios, G., and Mnih, A. The lipschitz constant of self-attention, 2020.

Kingma, D. P. and Dhariwal, P. Glow: Generative flow with invertible 1x1 convolutions. *arXiv preprint arXiv:1807.03039*, 2018.

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

Kosaian, J., Rashmi, K., and Venkataraman, S. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 30–46, 2019.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images (technical report). 2009.

Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., and Ramchandran, K. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2017.

Li, S., Maddah-Ali, M. A., and Avestimehr, A. S. A unified coding framework for distributed computing with straggling servers. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6. IEEE, 2016.

Liu, X., Gao, J., He, X., Deng, L., Duh, K., and Wang, Y.-Y. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. 2015.

Mao, X., Li, Q., Xie, H., Lau, R. Y., Wang, Z., and Paul Smolley, S. Least squares generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 2794–2802, 2017.

Marculescu, D., Stamoulis, D., and Cai, E. Hardware-aware machine learning: modeling and optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, 2018.

Mirza, M. and Osindero, S. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

Narra, K., Lin, Z., Kiamari, M., Avestimehr, S., and Annavaram, M. Distributed matrix multiplication using speed adaptive coding. *arXiv preprint arXiv:1904.07098*, 2019.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

Ruder, S. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

Song, Y., Meng, C., and Ermon, S. Mintnet: Building invertible neural networks with masked convolutions. In *Advances in Neural Information Processing Systems*, pp. 11004–11014, 2019.

Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 513–527, 2015.

Tandon, R., Lei, Q., Dimakis, A. G., and Karampatziakis, N. Gradient coding: Avoiding stragglers in distributed learning. In *International Conference on Machine Learning*, pp. 3368–3376. PMLR, 2017.

Verma, V., Lamb, A., Beckham, C., Najafi, A., Mitliagkas, I., Courville, A., Lopez-Paz, D., and Bengio, Y. Manifold mixup: Better representations by interpolating hidden states. *arXiv preprint arXiv:1806.05236*, 2018.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Yu, Q., Li, S., Raviv, N., Kalan, S. M. M., Soltanolkotabi, M., and Avestimehr, S. A. Lagrange coded computing: Optimal design for resiliency, security, and privacy. volume 89 of *Proceedings of Machine Learning Research*, pp. 1215–1225. PMLR, 16–18 Apr 2019. URL http://proceedings.mlr.press/v89/yu19b.html.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

Zhang, C., Patras, P., and Haddadi, H. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(3):2224–2287, 2019.

Zhang, H., Cisse, M., Dauphin, Y. N., and Lopez-Paz, D. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.