High-Ratio Lossy Compression: Exploring the Autoencoder to Compress Scientific Data

Tong Liu, Jinzhen Wang, Qing Liu, Shakeel Alibhai, Tao Lu, Xubin He, *Senior Member, IEEE*

Abstract—Scientific simulations on high-performance computing (HPC) systems can generate large amounts of floating-point data per run. To mitigate the data storage bottleneck and lower the data volume, it is common for floating-point compressors to be employed. As compared to lossless compressors, lossy compressors, such as SZ and ZFP, can reduce data volume more aggressively while maintaining the usefulness of the data. However, a reduction ratio of more than two orders of magnitude is almost impossible without seriously distorting the data. In deep learning, the autoencoder technique has shown great potential for data compression, in particular with images. Whether the autoencoder can deliver similar performance on scientific data, however, is unknown. In this paper, we for the first time conduct a comprehensive study on the use of autoencoders to compress real-world scientific data and illustrate several key findings on using autoencoders for scientific data reduction. We implement an autoencoder-based compression prototype to reduce floating-point data. Our study shows that the out-of-the-box implementation needs to be further tuned in order to achieve high compression ratios and satisfactory error bounds. Our evaluation results show that, for most of the test datasets, the tuned autoencoder outperforms SZ by up to 4X, and ZFP by up to 50X in compression ratios, respectively. Our practices and lessons learned in this work can direct future optimizations for using autoencoders to compress scientific data.

Index Terms—Lossy data compression, autoencoder, machine learning, scientific data

1 Introduction

Due to the information explosion in recent years, data reduction techniques such as data deduplication and lossless/lossy data compression are being widely used in storage systems to lower the cost of storage and I/O. In particular, data deduplication [1] is a special data reduction technique that eliminates duplicate copies of data. Lossless compression, meanwhile, refers to a class of data compression algorithms in which the original data can be exactly reconstructed from the compressed data. These include the well-known ZIP [2], gzip [3], FPC [4], and LZ4 [5]. According to recent reports [6], [7], although higher potential (e.g. 16x in some cases) exists, most practical lossless compression techniques on general benchmarks achieve a compression ratio of around 2x, with some up to 4x. While deduplication and lossless compression can perfectly reconstruct the original data without any data loss, they can only achieve relatively low compression ratios. On the other hand, lossy data compression schemes, at the expense of losing some data accuracy, usually achieve a higher compression ratio of 200x to 500x [8] and can even reach up to 10,000x [9]. In scenarios where file sizes can be significantly reduced before degradation is noticed by end users (such as in the case of multimedia data), lossy compression is usually the best choice. Popular lossy compression techniques include JPEG for images [10], MPEG for videos [11], and MP3 for audio [12].

- *T. Liu, S. Alibhai, and X. He are with the Department of Computer and Information Sciences at Temple University, Philadelphia, PA, 19122. E-mail: {tongliu, shakeel.alibhai, xubin.he}@temple.edu*
- *J. Wang, Q. Liu and T. Lu are with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ, 07102. E-mail: {jw447, qing.liu, lut}@njit.edu*

It has been reported [13], [14], [15], [16] that HPC scientific simulations can generate terabytes or even petabytes of data per run. In this case, deduplication and lossless compressors may not serve as good choices due to their relatively low compression ratios. It has been indicated [17] that deduplication can only achieve compression ratios of around $1.5 \sim 3$ on HPC data. Lossless compressors cannot achieve a much better result either: the lossless compressor FPC [4], for example, which is specifically designed to compress scientific data, can only reach a compression ratio of around 15x. Lossy compressors' high compression ratios, conversely, make them a much better solution to deal with the storage overhead challenge in HPC storage systems. However, because lossy compressors usually include a degree of data loss, lossy compression cannot be applied to cases where it is important that the original and decompressed data be identical or where huge deviations from the original data would be unfavorable. For HPC data compression, some data loss is acceptable; therefore, besides lossless compression methods [18], [19], lossy compression of HPC data has thus attracted a great deal of attention from researchers [20], [21], [22], [23], [24]. The data loss must be strictly contained within an error bound, however; to that end, some lossy, error-bounded compressors for floating-point scientific data, such as ISABELA [25], ZFP [26], and SZ [27], have recently been proposed for HPC scientific data compression.

An autoencoder [29] is a type of unsupervised artificial neural network commonly used to learn efficient data features. By taking advantage of its ability to conduct dimension reduction, several works have applied autoencoders to various forms of lossy data compression, such as image compression [30], [31], quantum data compression [32], and biometric patterns compression [33]. These works indicate

TABLE 1: Typical compressor/algorithm description

Compressor	Lossless/Lossy	Principle/Algorithm	Typical compression ratio
Deduplication	Lossless	Eliminate duplicate data chunks	$10 \sim 20$, $1.5 \sim 3$ for scientific data [17]
gzip	Lossless	DEFLATE algorithm	$1.5 \sim 2$ for scientific data [28]
FPC	Lossless	Sequentially predicting	$1.2 \sim 15$ for scientific data [4]
SZ	Lossy	Curve fitting model	$3.3 \sim 436$ for scientific data [27], [28]
ZFP	Lossy	Block transform	$3.2 \sim 226$ for scientific data [26], [28]
ISABELA	Lossy	Curve fitting, sorting	$2.1 \sim 88$ for scientific data [25], [28]

that, in certain scenarios, autoencoders possess the features of a lossy compressor. For example, Sento [31] shows that by using an autoencoder for image compression, image dimensionality can be reduced by around 60x. However, none of the previous work has explored how autoencoders work as a lossy compressor for HPC scientific data. In this paper, we conduct a comprehensive study on applying autoencoders to compress real-world HPC scientific data and aim to give guidance on how to use autoencoders from a user's perspective.

The contributions of this paper include:

1) To the best of our knowledge, we are the first to conduct a comprehensive evaluation of using an autoencoder as a lossy compressor for scientific data.
2) We perform experiments on 18 real-world HPC scientific datasets and show that, after our tuning operations, for most of the datasets, the autoencoder can achieve much higher compression ratios than the state-of-art HPC lossy compressors SZ (up to 4X) and ZFP (up to 50X) under common relative error-bounds.
3) We present observations and findings to help users better understand the compression autoencoder and use it wisely.

The rest of the paper is organized as follows. In Section 2, we describe the background and our motivation. In Section 3, we introduce our compression autoencoder prototype for scientific data and conduct preliminary experiments on it. In Section 4, we add our tuning methods and then present the compression ratio evaluation results after tuning. In Section 5, we report several observations and offer guidance for using an autoencoder as a lossy compressor. In Section 6, we discuss related work. The conclusion is given in Section 7.

2 Background and Motivation

To meet the steadily increasing demand for HPC data storage, many lossy and lossless compressors and algorithms have been applied in practice. Table 1 shows the typical compression ratios of several commonly-used lossless and lossy scientific data compressors and algorithms. As mentioned in Section 1, HPC scientific simulations can produce terabytes or even petabytes of data per run, which is a huge challenge for storage systems. A $1.5 \sim 15$ compression ratio is not enough to relieve this storage stress. Although lossy compressors such as SZ, ZFP, and ISABELA can sometimes reach compression ratios in the hundreds, previous research [28] has found that, under reasonable error bounds, it is hard for these compressors to achieve a compression ratio up to 100x in general cases. As a result, there exists an urgent need to find a compressor with a much higher compression ratio while still satisfying the compression requirements of HPC scientific data.

An autoencoder is a popular type of neural network commonly used for feature learning and dimension reduction. The simplest form of an autoencoder is a feedforward, non-recurrent neural network that aims to copy its inputs to its outputs. The structure of an autoencoder with three fully-connected hidden layers is shown in Figure 1. In this simple example, there are three layers L_1, L_2, and L_3, which represent the input, an intermediate hidden layer, and the output, respectively. In addition, an autoencoder always consists of two parts, the encoder and decoder, which can be defined as transitions ϕ and ψ, respectively. Assuming we have a set of training examples $X = \{x_1, x_2, x_3, x_4, x_5\}$, a set of code layer neurons $Z = \{z_1, z_2, z_3\}$, and a set of trained output as $X^{'} = \{x_1^{'}, x_2^{'}, x_3^{'}, x_4^{'}, x_5^{'}\}$, then we have:

$$\phi : X \to Z \tag{1}$$

$$\psi : Z \to X^{'} \tag{2}$$

In this simple example, there is only one hidden layer, so we have:

$$Z = \sigma(WX + b) \tag{3}$$

where σ is an element-wise activation function, such as a sigmoid function or a rectified linear unit; W is a weight matrix; and b is a bias vector. The Z layer is usually referred to as a code layer, which can be regarded as a compressed representation of the input X. After encoding, we then have the following mapping from Z to the reconstruction $X^{'}$, which is the same shape as the input X:

$$X^{'} = \sigma^{'}(W^{'}Z + b^{'}) \tag{4}$$

Essentially, we want the output to be equal to the input: $X^{'} = X$. However, due to the inner property of the neural network, there are almost always some data loss during the reconstruction. Usually, an autoencoder's reconstruction error (also known as a cost function) is defined as squared errors:

$$J = \|X - X^{'}\|^2 = \sum_{n=1}^{N} \|x_n - x_n^{'}\|^2 \tag{5}$$

In the case where there are fewer hidden units m in Layer L_2 than input units n in Layer L_1, the network is forced to learn a compressed representation of the input. Thus, a compression autoencoder (CAE) is formed. For example, if some of the input features are correlated, then the CAE is able to learn those correlations and reconstruct the input data from a compressed representation. This implies that the CAE has excellent potential to be applied for data compression. Several works have applied CAEs in the lossy data compression field (such as for image compression [30], quantum data compression [32], and biometric patterns

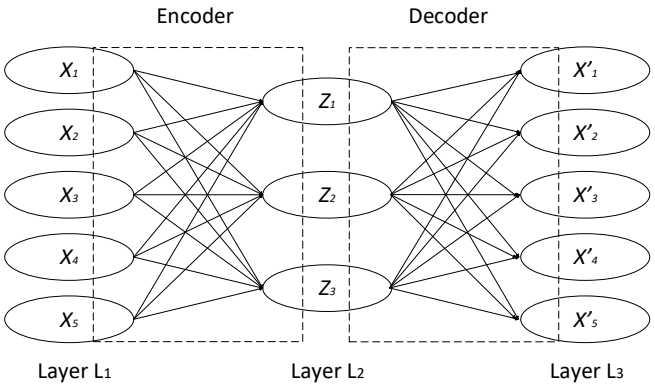


Fig. 1: A fully-connected three-layer autoencoder.

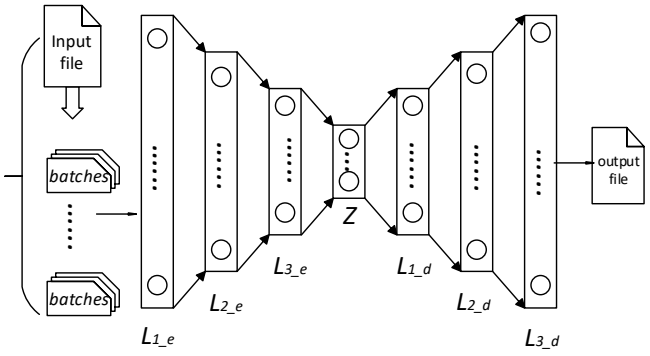


Fig. 2: The seven-layer compression autoencoder.

compression [33]), and they all achieve promising performance. From this, we believe that autoencoders may also perform well as a lossy compressor for HPC scientific data, providing a much higher compression ratio than existing lossy compressors. To test this, we implement and configure a prototype autoencoder, as discussed below.

3 Preliminary Evaluation

In this section, we first give a detailed introduction to the autoencoder prototype we will use in the evaluation. Then, based on the autoencoder prototype, we configure a CAE specifically used for scientific data lossy compression. Finally, we introduce the HPC scientific datasets we use in this work and give some preliminary experiment results.

3.1 CAE Prototype Implementation

To evaluate an autoencoder's compression performance on HPC scientific data, we implement an autoencoder prototype specifically designed for large-scale, double-precision floating-point data based on the open-source TensorFlow [34] machine learning framework. As shown in Figure 2, the autoencoder has seven layers with three layers $L_{1\_e}, L_{2\_e}, L_{3\_e}$ in the encoder part, three layers $L_{1\_d}, L_{2\_d}, L_{3\_d}$ in the decoder part, and one code layer Z. Before the input file I (which contains the scientific data) enters the neural network, it is divided into several batches $b_i \in I$. After the input is divided into batches, each batch b_i, containing a part of the original scientific data, will go into the input layer L_{1_e}. When a batch enters the input layer, each element of the original scientific data becomes one neuron in the layer. Each point in each batch contains x original data points, where x is the number of neurons in L_{1_e}. In the encoder part, the number of neurons decreases as the layer goes from the input layer L_{1_e} to the code layer Z. Each layer in the encoder part has a weight matrix and bias vector that are used to accomplish dimension reduction. After three layers of compression, the information stored in L_{1_e} is represented in layer Z with significantly fewer neurons. The decoder is similar to the encoder in that each layer has a weight matrix and bias vector. The information in the Z layer goes through the three decoder layers and is then written to the output file. If the whole autoencoder is regarded as a compressor, then layer L_{1_e} is the original file, layer Z is the compressed file, layer L_{3_d} is the decompressed file, and the encoder and decoder represent compression and decompression, respectively. More details of the implementation can be found at the link provided in Section 7.

Assume the size of the input file is S_i, the sum of all the Z layer sizes are S_z, and there are N numbers in the input dataset X and the output dataset $X^{'}$ (as they should have the same size). The theoretical compression ratio, CR_{th}, would then be:

$$CR_{th} = \frac{S_i}{S_z} \tag{6}$$

If any other files (having a combined size F) need to be stored to assist in the decompression process, then the actual compression ratio, CR_{ac}, would be defined as:

$$CR_{ac} = \frac{S_i}{S_z + F} \tag{7}$$

In addition, let the point-wise error for each input data point be e_i. Then the average point-wise relative error E, which usually serves as the measurement criteria of an error-bounded compressor, will be:[1]

$$E = \frac{\sum_{n=1}^{N} e_i}{N} = \frac{\sum_{n=1}^{N} \frac{\left|x_i - x_i^{'}\right|}{x_i}}{N} \tag{8}$$

In the following evaluation part, these metrics are used as the criteria for the lossy compression performance.

3.2 CAE for Scientific Data

In this paper, we aim to evaluate the performance of autoencoders as lossy compressors for HPC scientific data. All experiments are conducted on a server running Ubuntu 16.04.5 LTS with an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 32 GB of memory (clock: 2133MHz). We also use the online Google Colaboratory environment [35] for some preliminary experiments. The datasets we use for testing come from open-source HPC scientific data benchmarks,

[1]If the original value is zero, then we do not apply the error formula to the predicted value, as the point-wise relative error function on a zero value would be undefined. However, the zero values will still be error-bounded by our delta-value and delta-index mechanism proposed in Section 4.

TABLE 2: Detailed descriptions of the HPC scientific datasets

Dataset	Benchmark	Size	Description
blast2	FLASH	4631KB	Strong shocks and narrow features
maclaurin_p	FLASH	2134KB	MacLaurin spheroid pressure (gravitational potential at the surface/inside a spheroid)
maclaurin_t	FLASH	2134KB	MacLaurin spheroid temperature (gravitational potential at the surface/inside a spheroid)
sedov	FLASH	625KB	Hydrodynamical test code involving strong shocks and non-planar symmetry
md_seg	GROMACS	4800KB	Molecular dynamics simulation of Lysozymein protein in water
s3d_p	MCERI	1552KB	A 3D streamline simulator for tracer flow and two-phase displacement
2d_annulus	NEK5k	2910KB	Simulation of natural convection between two concentric cylinders
astro	NEK5k	524KB	Velocity magnitude in a supernova simulation
bump	NEK5k	400KB	Flow density of an axisymmetric bump
eddy	NEK5k	2160KB	2D solution to Navier-Stokes equations with an additional translational velocity
fish	NEK5k	524KB	Velocity magnitude in a CFD calculation of cooling air being injected into a mixing tank
rarefaction	NEK5k	1600KB	Simulation of a two-dimensional box with walls on all sides and a subsonic outflow
swept	NEK5k	1234KB	Swept particles in a triply periodic 3D domain
vortex	NEK5k	1600KB	Inviscid Vortex Propagation: tests the problem in earlier studies of finite volume methods
yf17_p	NEK5k	776KB	Pressure in a computational fluid dynamics calculation
yf17_t	NEK5k	776KB	Temperature in a computational fluid dynamics calculation

such as NEK5000, MCERI, GROMACS, and FLASH[2]. Detailed descriptions of the datasets are presented in Table 2.

After generating the scientific data, we convert it into double-precision floating-point data (8 bytes per number) and then write the data to a binary file. On average, each binary file has around 400,000 data numbers. We divide this binary file into two equal parts. The first part serves as the training dataset, which goes to the encoder part of the autoencoder for training over multiple epochs. During each epoch, the weight matrices and bias vectors are updated by the optimizer. After the entire training process completes, we are left with the final weight matrices and bias vectors. Then the second part of the binary file, which serves as the testing dataset, enters the encoder part of the autoencoder. It uses the final weight matrices and bias vectors generated from the training step to proceed through the autoencoder's encoder and decoder. If we consider the testing step as the compression process, then we can think of the testing input file as the original data file which needs to be compressed, the encoder part of the testing step as the compression process, the Z layer as the compressed file, the decoder part of the testing step as the decompression process, and the testing output file as the decompressed file. What we mainly focus on in the autoencoder is the testing prediction error, which displays the prediction accuracy of the autoencoder.

Within the autoencoder, there are several important parameters that we can configure.

- *Number of layers:* The number of layers in a neural network could have a significant impact on the training performance. In order to find a suitable number of layers for our autoencoder prototype, we choose, without loss of generality, three datasets, *yf17_t*, *maclaurin-p*, and *swept*, to test their actual compression ratios under different numbers of layers with error bounds 0.1, 0.01, and 0.001. We tested the datasets with a theoretical compression ratio of 512; the results (provided at this link as well as in the supplemental document) indicate that, under all error bounds, a seven-layer autoencoder is the best option in general. More layers may give worse performance because of the *overfitting* problem [36]: the risk that, when we try too hard to find the very best model for the training data, we end up fitting to the noise in the training dataset by memorizing its various peculiarities rather than finding a general predictive rule.
- *Number of neurons in each layer*: Autoencoders can achieve different theoretical compression ratios depending on the number of neurons in each layer. The theoretical compression ratio between two consecutive layers, such as L_{1_e} and L_{2_e}, is the quotient of their neuron numbers, $n_{L_{1\_e}}/n_{L_{2\_e}}$. The theoretical compression ratio determines the strength of the dimension reduction.
- *Batch size:* The batch size defines the number of samples that are propagated through the network. In our tests, we split each input file into multiple batches (normally 64) before it enters the encoder part of the autoencoder.
- *Training epochs*: The number of training epochs determines how many times a training dataset will be trained by the neural network.
- *Loss function:* An important part of an artificial neural network is its loss function, which is used to measure the inconsistency between the predicted and actual values. The original loss function is $loss\_original = tf.reduce\_mean(tf.pow(prediction - true, 2))$. To ensure that the loss function we choose does not negatively impact the compressor performance, we select another three loss functions for comparison. Without loss of generality, we choose three datasets, *yf17_t*, *maclaurin-p*, and *swept*,

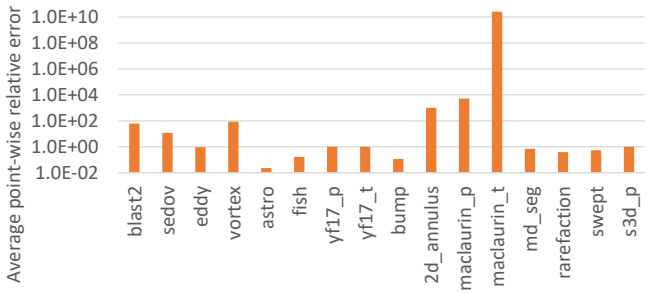


Fig. 3: The raw performance regarding the average point-wise relative error of the autoencoder under a theoretical compression ratio of 512 on a logarithmic scale.

[2]Nek5000: https://nek5000.mcs.anl.gov/files/2015/09/NEK_doc.pdf;
MCERI: http://www.pe.tamu.edu/mceri/software.html;
GROMACS: http://www.gromacs.org/About_Gromacs/Benchmarks;
FLASH: http://flash.uchicago.edu/site/flashcode/user_support/

with theoretical compression ratios of 512 to test the actual compression ratios under different loss functions with error bounds 0.1, 0.01, and 0.001. The detailed results (provided at this link as well as in the supplemental document) clearly show there is no significant difference among the various loss functions.

To test the raw compression performance with respect to the prediction error of the autoencoder, we use the compression model to conduct preliminary experiments on the 16 HPC datasets listed in Table 2 under a theoretical compression ratio of 512. The results, shown in Figure 3, indicate that if an input file is directly put into the autoencoder, the testing error could be as high as 1.0E+10; this is not acceptable for HPC scientific data compression. We also studied the activation functions and optimizers in TensorFlow, but found that none of these significantly help. Thus, in order to make the autoencoder practical for HPC data compression, we must explore some other methods to improve the prediction accuracy.

Finding 1: ***Simply using an autoencoder out-of-the-box to compress HPC scientific data may lead to unacceptably high compression error, and merely tuning the prototype parameters cannot rectify this problem.***

4 Tuning the Autoencoder for Scientific Data Compression

In this section, we first propose two prediction accuracy tuning schemes which make the autoencoder satisfy the error-bound requirements for HPC data compression. We then propose two storage-reduction tuning schemes which improve the compression ratio of the autoencoder.

4.1 Prediction Accuracy

4.1.1 Data normalization

After examining the original datasets, we find that different HPC scientific datasets may have numbers in very different orders of magnitude; normally, they could be anywhere from $10^{-8} \sim 10^{8}$. When these numbers proceed through the autoencoder, they go through the sigmoid activation function in both the encoder and the decoder parts. The sigmoid function maps all the numbers into the range $(0, 1)$. In order to make datasets with different magnitudes suitable for the autoencoder, we implement a data normalization scheme. Before an input file enters the autoencoder, we map all numbers into the range $[0.01, 0.1)$.[3]We store the normalized numbers N_n and the corresponding exponential numbers N_e. The normalized numbers then go into the encoder part of the autoencoder. When we need to decompress the datasets, we use the corresponding exponential numbers to convert the numbers back to their original magnitude before writing the final output. To test the performance of this data normalization scheme, we implement it in the prototype and test all 16 datasets we have in Table 2.

Figure 4 shows the average point-wise relative testing errors for the 16 experimental datasets with and without data normalization. This figure indicates that, after adopting this scheme, all the average point-wise relative errors for these datasets are under 1.00, and most are under 0.1. This makes the autoencoder practical for compressing HPC scientific data.

4.1.2 Delta numbers

Simply having a low average prediction error is not enough to make autoencoders applicable for lossy compression on HPC scientific data; we also need to have the autoencoder error-bounded. For example, if the average prediction error of a compression test is 0.05, and the error bound requirement E_{eb} is 0.1, then it could be the case 80% of the output numbers have a testing error of 0.01, but the rest have testing errors as high as 200%, if not higher. To solve this problem, we examine every predicted number N_i. If its prediction e_i is lower than E_{eb}, then nothing will be done; otherwise, if e_i is higher than E_{eb}, we store the difference between the input number and the output number as the delta number d_i and the index for this number i. When we need to decompress the file, we read the delta number files and the index files to add the delta numbers to the predicted values. The delta numbers are sequentially matched to the delta indices. If x is the position in the delta number file that a delta number n appears at, then the index of n in the original dataset is the value in position x in the delta index file. By adding the delta numbers to the predicted values, the error bound will be met.

Finding 2: ***Normalizing the data and saving the delta values can significantly reduce the prediction error and enable the autoencoder to be error-bounded, thus making it practical for compressing HPC scientific data.***

4.2 Storage Overhead

By storing the delta numbers, the error bound requirement is met. However, this involves storing extra files containing the delta numbers and the indices of the delta numbers. Sometimes these files can be extremely large; they may even be several hundred times the size of the compressed Z file. The main storage overhead comes from two files: the *delta-value* file f_{dv}, which stores the deltas (the differences between the input and output numbers), and the *delta-index* file f_{di}, which stores the indices of the numbers that have a delta number stored.

- The *delta-value* file f_{dv} stores the difference d_i between the number in the input file and the predicted value for the numbers whose prediction errors are greater than the error bound. To reduce the size of f_{dv}, we employ two solutions. 1) For the input and output files, if the numbers are stored as double-precision floating-point data, each number will be 8 bytes. As these delta values are meant to correct the predicted data, their first $2 \sim 3$ digits are typically the most important. As a result, instead of using

[3]Note that if an input value is 0, then it would not be normalized into the specified range; the range is thus more accurately described as 0 U [0.01, 0.1). In addition, a different order of magnitude (e.g. [0.1, 1)) may be used, as long as it is in the range (0, 1). Choosing a different order of magnitude, however, may affect the accuracy of the autoencoder.

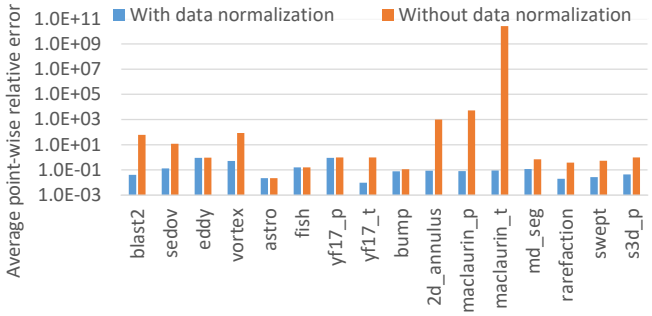


Fig. 4: Comparison of the average point-wise relative errors with and without the data normalization scheme on a logarithmic scale.

double-precision, we use half-precision to store these delta numbers, thus reducing the size of each number to 2 bytes. Therefore, the *delta-value* file size is one-fourth its original size. 2) We can also use lossy compression to further compress the file. If we use the SZ compressor to compress the delta numbers with an error-bound of 0.1, they can be further compressed by $10 \sim 20$ times.[4]

- The *delta-index* file stores the indices of the numbers whose prediction errors are higher than the error bound. Unlike the *delta-value* file, the indices have to be completely correct after decompression, because we have to add the delta values to the correct input numbers. To reduce the size of the *delta-index* file, we also adopt two applications: 1) Instead of storing every index number, we implement a bitmap to store, for every number, either a 1 (indicating a delta correction) or a 0 (indicating no delta correction). Thus, for each double-precision number, we only need 1 bit to indicate whether or not it needs delta correction. 2) After the bitmap file is generated, we further compress it using the lossless compressor bzip2 [37]. By applying the bzip2 compression, the bitmap can be further compressed by $5 \sim 20$ times.

After applying these storage optimization schemes, we conduct experiments on the 16 datasets to compare the compression performance of the autoencoder with the state-of-the-art error-bounded HPC scientific data lossy compressors SZ and ZFP. The two metrics we mainly focus on in the comparison tests are the compression ratio and the error bound.

4.3 Compression Ratio Comparison After Tuning

4.3.1 Compression ratio calculation

In general, the compression ratio is defined as the ratio of the original dataset size to the output (compressed) file size. This is how SZ and ZFP calculate compression ratios. Calculating the compression ratio for the autoencoder is slightly more complicated, as there is extra metadata that needs to be stored in addition to the compressed file. In this paper, we use two types of compression ratios (as described in Section 3.2): the theoretical compression ratio and the actual compression ratio. All the compression ratios we will compare with SZ and ZFP are the actual compression ratios that an autoencoder can achieve. When calculating the actual compression ratio of the autoencoder, the output file size is the sum of the compressed file (the Z layer), the data normalization metadata, the weight and bias metadata, and the error-bound metadata (the *delta-index* and *delta-value* files). For the data normalization metadata, since most of the numbers are in the same or close magnitudes in most of the HPC scientific datasets, we only store the exponential numbers and the number of sequential data points that have the same exponent. This is usually only around 10 bytes.

For all the experiments in this work, we use the same neural network (the seven-layer autoencoder) to compress the datasets. The size of the weight and bias metadata files only depend on the neural network and is fixed if the same neural network is used. This is because the metadata saves the graph's weight and bias information so that, after the training process, they can be restored and used to perform compression or decompression. In our case, the combined size of these files is 272KB.

There are two important points to note here. First, in this paper, we use a total of 18 real-world datasets. As listed in Table 2, 16 of them have small sizes (i.e, less than 10MB); the two large datasets are discussed separately in Section 5.2. Second, we do not include the weight and bias metadata files when calculating the actual compression ratios of the small datasets. The first point is a result of most of our available datasets being small in size, as well as the fact that small datasets generally take a more acceptable amount of time to train in our current experimental environment. More importantly, the size of the file to be compressed is not the determining factor of the compression ratio; rather, the data features (such as the value distribution) play a larger role. As for the second point: in Section 5.2, we conduct tests on large datasets to show that when the dataset size is large (i.e. over 1000MB), which is more practical for HPC datasets, the weight and bias metadata files' impact on the actual compression ratio are negligible. It is fine to disregard the weight and bias files here, on small datasets, because these compression ratios are meant to be representative of the compression ratios we would get on larger files. Since we can get the same compression ratios (excluding weights and biases) on small and large datasets, our compression ratios on small datasets (without weights and biases) are very similar to what we would get in the real world on large datasets (with weights and biases included).[5]

Another factor impacting the compression ratio is the error bound. The error bound limits the accuracy loss during compression. There are two commonly used error bound measurements: absolute error and relative error. Assume a data point from the input file is n_i and the error bound is set as α. Then a point-wise absolute error bound means the

[4]Based on our experiments, the predicted values are always close to the original values, which implies that the delta value is always smaller than the original value. Thus, further compression of the delta value will not make the final predicted number exceed the error bound. However, there may exist extreme cases in which the delta value may be much higher than the original value. This rare case may occur if the dataset is considered an CAE-bad dataset, as discussed in Section 5.1.

[5]It is also relevant to mention that many related works [27], [28] also use these small datasets for testing, so using them here as well makes comparisons easier and more reasonable.

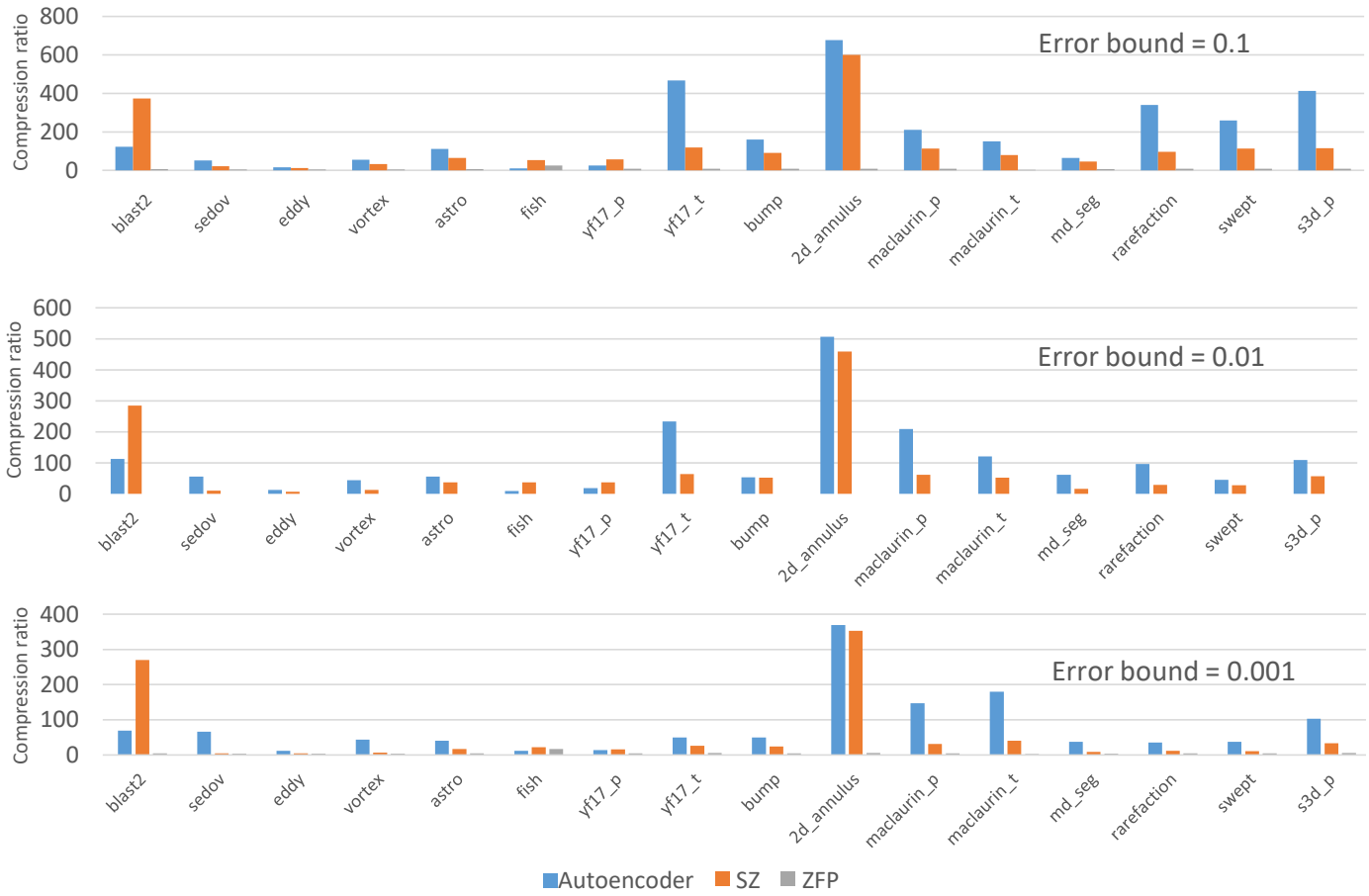


Fig. 5: The comparison of the compression ratios among the autoencoder, SZ, and ZFP compressors under the same three error bounds (0.1, 0.01, and 0.001).

decompressed data point $n_i^{'}$ will be in the range $[n_i - \alpha, n_i + \alpha]$, and a point-wise relative error bound means $n_i^{'}$ will be in the range $[n_i \cdot (1-\alpha), n_i \cdot (1+\alpha)]$. We use point-wise relative error when testing the autoencoder, SZ, and ZFP, which is a commonly used metric in related HPC lossy compression works [38], [39], [40].

4.3.2 Result comparison

By setting the number of neurons in each layer of the autoencoder, we can determine the theoretical compression ratio. The prediction error, on the other hand, will be determined after the compression process. For the SZ and ZFP compressors, the error bound will be given before the compression as one parameter and the compression ratio will be calculated after the compression process. To ensure a fair comparison with respect to compression performance, we implement the error-bounded function in the autoencoder so that we can set a common error bound for the autoencoder, SZ, and ZFP and then compare their respective compression ratios.

Due to the properties of the autoencoder, a very strict error-bound may not be practical now; that is one of our future research interests. In addition, for the autoencoder, we need to first set a theoretical compression ratio; the actual compression ratio will then be calculated based on the prediction performance. However, as we will discuss in Section 5.3, there is no clear relationship between the theoretical compression ratio and the prediction performance. As a result, in this comparison we choose three error-bounds, 0.1, 0.01, and 0.001, and in the autoencoder prototype, we set the theoretical compression ratio to 512 for most of the datasets. (For SZ and ZFP, a compression ratio of 512 is usually hard to achieve with an error bound of 0.1 to 0.001.) The one exception to this is the *2d_annulus* dataset: as it can achieve a compression ratio of 600.06 with an error bound of 0.1 on the SZ compressor, we set its theoretical compression ratio on the autoencoder to 2048 in order to make it possible for the autoencoder to outperform SZ. Figure 5 shows the comparison results.

This figure indicates that, for most of these test datasets (13 out of 16), the autoencoder has a higher compression ratio than SZ and ZFP compressors under these three common error-bound requirements. The best improvement comes from the *yf17_t* dataset: the autoencoder achieves a compression ratio of 467.47, 3.92x the SZ compression ratio of 119.15. As for ZFP: since its main advantage is high compression throughput and strict error-bound, and since it can only take absolute error bound as input, it has a much lower compression ratio than the autoencoder and SZ under the same point-wise relative error bound in most cases. Since we set most of the theoretical compression ratios to 512 for this experiment, some of the datasets actually do not yet achieve their highest possible compression ratios. As we will see in Section 5.3, there is a trade-off between the

compression ratio and the prediction accuracy. By tuning the parameters of the autoencoder prototype, there exists a high probability that some of the datasets can achieve an even higher compression ratio under the demanded error-bound requirement. Since the tuning schemes are particularly focused on each data point and have no dependency on the original file size, there is no scalability issue with the CAE. As a result, when the original file size is much larger, the CAE can still provide similar compression performance.

Finding 3: ***For most of the real-world scientific datasets (13 out of 16), the tuned autoencoder outperforms the compression ratios of SZ by 2 to 4X, and ZFP by 10 to 50X, respectively, under common error bounds.***

4.4 A Case Study on CAE

In addition to the direct comparison of compression ratios, we also conduct experiments to evaluate the CAE's performance with respect to the application error. CGNS[6]provides the unstructured mesh for the Northrop YF-17 fighter aircraft so that we can use the dataset *yf17_t* for the visualization application tests. In this test, we choose three actual compression ratios, 30x, 50x, and 100x, and then examine the visualization performance of the CAE and SZ. Figure 6 indicates that the CAE and SZ can both almost perfectly reconstruct the mesh topology of the dataset under a compression ratio of 30x. When the compression ratio increases to 50x and 100x, the CAE can still maintain most of the features of the mesh topology. With respect to SZ, however, the visualization becomes distorted when the compression ratio reaches 50x and 100x. This evaluation validates our theory that the autoencoder technique is promising to compress HPC data by two orders of magnitude while still maintaining the usefulness of the data.

5 Observation

From the previous section, we notice that while our autoencoder provides a higher compression ratio than SZ and ZFP under a common error-bound for most of the datasets, some of the datasets perform worse. In this section, we first attempt to explore the inner data features of the experimental datasets and aim to find a rule that can be referenced for when an autoencoder is used for compressing HPC scientific data. We then offer guidance on the relationships among the compression metrics of the autoencoder.

5.1 Data Features of Experimental Datasets

To discover which data features are good indicators for lossy compression via an autoencoder, we look at the following metrics on the datasets:

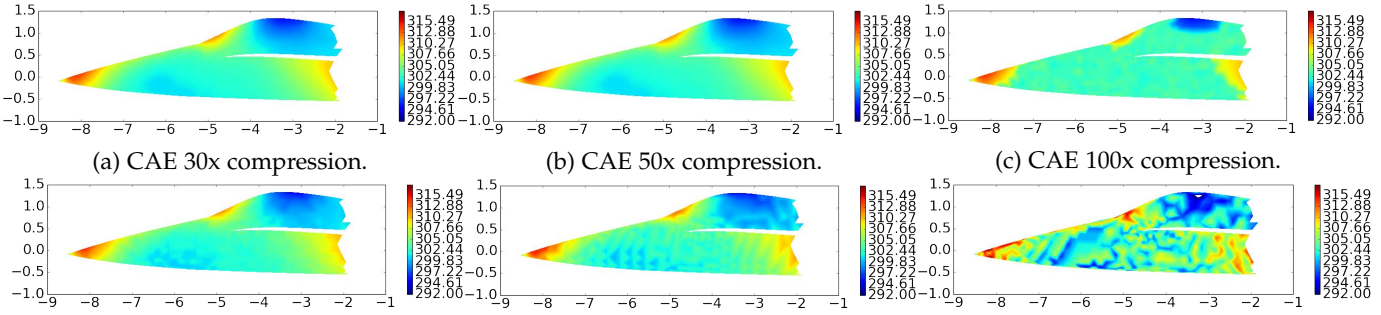
- *Cumulative distribution function*: The cumulative distribution function (CDF) curve shows the distribution of numbers in a dataset, as well as the range and skewness of the dataset.
- *Byte entropy*: Byte entropy is the number of bits per character. It is in the range [0, 8] and is used to gauge the information density of dataset content. The higher the information density is, the lower the compressibility is.
- *Coreset size*: Coreset size is the number of unique symbols (bytes in our case) that compose the majority of a dataset (e.g., 90% of all symbols). A small coreset size means there exist considerable repeated symbols in a dataset. The range of the coreset size is [0, 8] on a logarithmic scale.
- *Serial correlation*: Serial correlation measures the extent to which each byte depends on the previous byte in a dataset. For random data, this value is expected to be close to zero; for highly correlated data, it approaches one. The theoretical values of serial correlation coefficients range in [-1, 1]. This metric may distinguish random data from data that bear patterns.
- *Coefficient of variation*: The coefficient of variation (CV), defined as the ratio of the standard deviation to the mean, measures the dispersion of a frequency or probability distribution. CVs are often expressed as percentages; the higher the percentage is, the higher the extent of variability relative to the mean of the population is.

We have listed all the datasets under two categories, CAE-good and CAE-bad. CAE-good means that, after all the tunings, the dataset can have a relatively good raw autoencoder prediction accuracy (average point-wise relative error under 10%) under a certain theoretical compression ratio. (In these tests, the theoretical compression ratio is 512.) CAE-bad means that the dataset has an unacceptable raw autoencoder prediction accuracy (average point-wise relative error greater than 10%). Figures 7 and 8 show the detailed data features of the 16 datasets. These two figures indicate that the CDF, byte entropy, coreset size, and serial correlation are not good indicators for autoencoder-suitable datasets. For a given dataset, each of these four metrics can be either high or low, regardless of whether that dataset belongs to the CAE-good or the CE-bad category. However, there is one metric, CV, which, in most of the cases, matches well with the autoencoder's prediction accuracies. If a dataset's CV is low (meaning the dataset is very smooth), then the autoencoder can produce a good error prediction for that dataset. If, however, the CV is high (indicating that the data may jump a lot), then the autoencoder's error predictions are poor for that dataset. From the experimental results, we can roughly estimate that if the CV is lower than 0.40, then, using a theoretical compression ratio of 512, our autoencoder can provide an average point-wise relative error bound lower than 0.1.

Finding 4: ***The coefficient of variation (CV) can, in most cases, serve as a good indicator to quickly determine whether a dataset is suitable for compression by an autoencoder. The lower the CV is (i.e. the smoother the dataset is), the better the autoencoder can perform.***

However, from these datasets, we can find an exception to this rule: *yf17_p*. This dataset has a bad autoencoder prediction error of 0.92, even though it has a very low

[6]CGNS: https://cgns.github.io/index.html



(a) CAE 30x compression. (b) CAE 50x compression. (c) CAE 100x compression.

(d) SZ 30x compression. (e) SZ 50x compression. (f) SZ 100x compression.

Fig. 6: Unstructured mesh visualization of the *yf17_t* dataset with CAE and SZ compression. ZFP results are not shown here because it cannot reach 30x, 50x, and 100x compression ratios for the *yf17_t* dataset. (These compression ratios are also hard to reach for any other typical dataset with ZFP.) The original *yf17_t* dataset visualization is provided at this link as well as in the supplemental document.

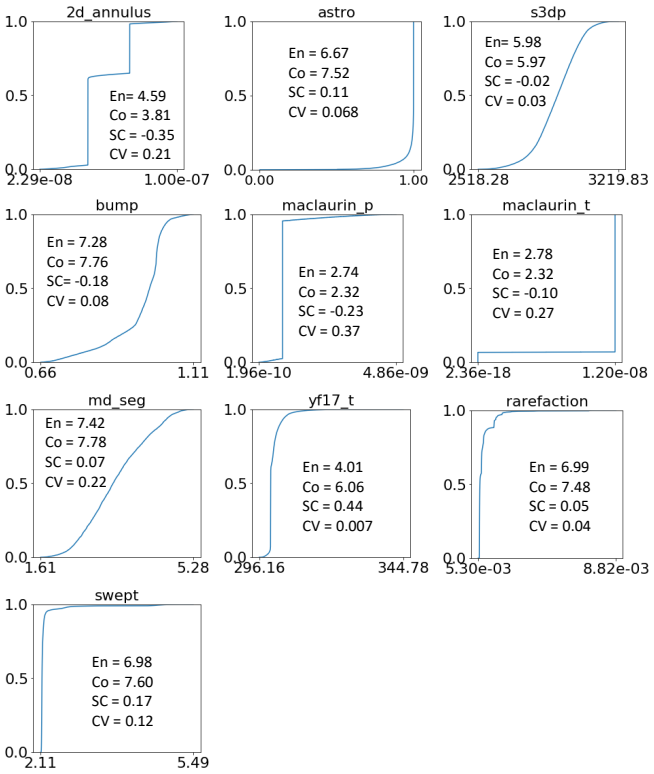


Fig. 7: The ten datasets with good CAE compression error predictions. The curves in the figures show the CDFs of data values. *En*, *Co*, *SC*, and *CV* stand for byte entropy, coreset size, serial correlation, and coefficient of variance, respectively.

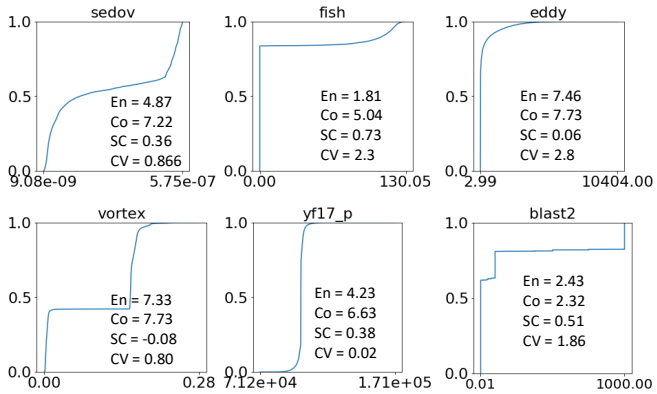


Fig. 8: The six datasets with bad CAE compression error predictions. The curves in the figures show the CDFs of data values. *En*, *Co*, *SC*, and *CV* stand for byte entropy, coreset size, serial correlation, and coefficient of variance, respectively.

CV of 0.02. This is because almost all the elements in this dataset hover around 100,000, and this could lead to a prediction problem for the autoencoder. By selecting a small part of this dataset, for example, we get four contiguous numbers {99528.03906, 99230.46875, 110569.96093, 109720.05468}. Directly looking at these four numbers may give the impression that this dataset is very smooth. However, after data normalization, these four numbers are normalized to {0.9952803906, 0.9923046875, 0.11056996093, 0.10972005468}. It is clear that the data smoothness is now seriously broken, as the numbers jump directly from 0.99 to 0.11. Within the entire dataset, this type of jump happens thousands of times, and thus the CV of the normalized numbers entering the encoder part of autoencoder is very high. A more accurate method to determine whether a dataset is good for compression via the autoencoder may thus be to calculate its CV after normalization; however, this would come with the additional overhead of normalizing the data first.

5.2 Impact of Weight and Bias Metadata on Compression Ratios

In the autoencoder prototype, we use TensorFlow's API *tf.train.Saver()* to save the weights and biases into three metadata files: *checkpoint*, *.ckpt.index*, and *.ckpt.data*. (There is another *.ckpt.meta* file that contains neural network information, but it is not necessary since that information is contained within the autoencoder code. TensorFlow includes an option to prevent this file from being saved.) In this paper, since all our tests are conducted with the same autoencoder neural network and a theoretical compression ratio of 512,

we have a fixed metadata file size, 272KB. In all previous tests, since the dataset sizes are small, we do not include the weight and bias metadata files with the compressed files. The purpose of those tests is to show that among multiple datasets, the autoencoder has a high possibility to achieve a higher compression ratio than SZ and ZFP. In this section, we will use both mathematical analysis and real dataset experiments to prove that when dataset size is large (i.e. over 1000MB), the weight and bias metadata files' impact is very small and can be ignored. We do not use large datasets for the experiments in Sections 3 and 4 because the training/testing process for such datasets usually takes a long time.

5.2.1 Analysis of dataset size impact

Assume the original data size is S_o, the compressed file size without WB (weight and bias) metadata files is S_c, and the WB metadata file size is S_{wb}. Then we have the compression ratio without WB metadata,

$$CR_1 = \frac{S_o}{S_c} \tag{9}$$

and the compression ratio with WB metadata,

$$CR_2 = \frac{S_o}{S_c + S_{wb}} \tag{10}$$

Since S_{wb} is fixed, as S_o and S_c becomes larger, CR_1 and CR_2 will become closer. Figure 9 shows the difference between CR_1 and CR_2 when the actual compression ratio is set as 100, 50, and 10 with error bounds 0.1, 0.01, and 0.001, respectively. The two curves merge quickly when the original dataset size is greater than 200MB.

5.2.2 Real-world dataset evaluation

We choose two large datasets from the *Scientific Data Reduction Benchmarks*[7], *SCALE-LETKF* and *HACC*, with file sizes 3.39GB and 1.69GB, respectively. Again, we compare the compression ratio of the autoencoder with SZ and ZFP under three error bounds: 0.1, 0.01, and 0.001. The results are shown in Figure 10. The figure indicates that, for large datasets, the autoencoder's compression ratios with and without the weight and bias metadata files are very close, and both of them show significant superiority over SZ and ZFP. These results match our previously reported results for the small datasets, on which the autoencoder produces compression ratios 2 to 4 times those of SZ and ZFP. We also show the breakdown of the compressed data for these two datasets in Figure 11 (error bound set to 0.001). The results indicate that the WB metadata (dark blue part on the top of each bar) only represents less than 3% of the total compressed data.

Our analysis and evaluation of the autoencoder's compression performance on large files confirm that when the dataset is big, the weight and bias metadata files have a negligible impact, and therefore it is reasonable to disregard them when performing the evaluation in Section 4.

[7]SDR Benchmarks: https://sdrbench.github.io/

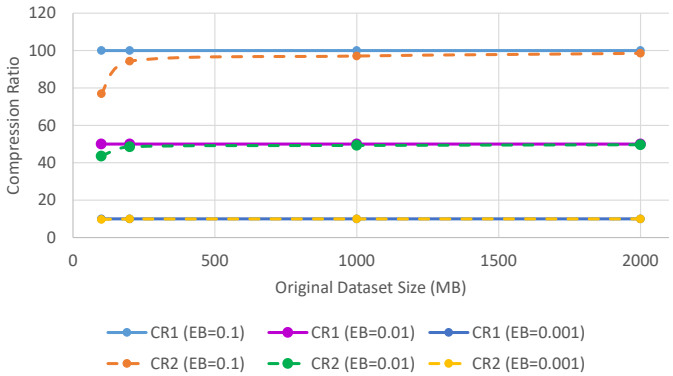
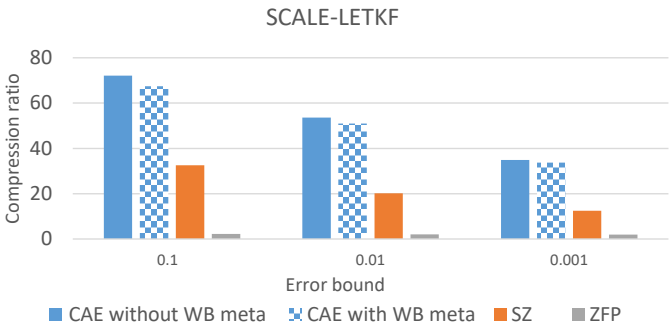
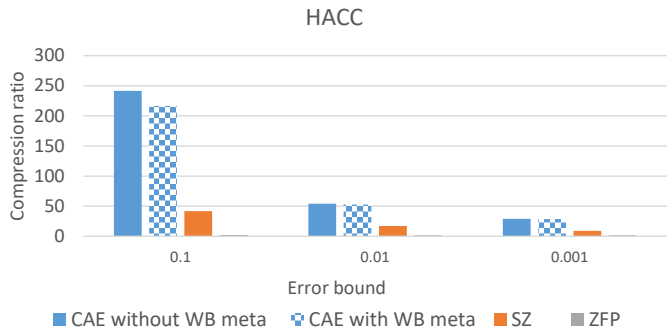


Fig. 9: The difference between the compression ratio with (dashed line) and without (solid line) WB metadata files as the original dataset size increases.



(a) SCALE-LETKF with 846,720,000 single-precision floating-point numbers.



(b) HACC with 421,430,799 single-precision floating-point numbers.

Fig. 10: Compression ratios with and without WB metadata files for two big datasets.

5.3 Prediction Accuracy Under Different Compression Ratios and Training Epochs

In Section 3.2, we briefly introduced the definitions of compression ratios and training epochs. We believe it would be helpful for users to know the relationships between the prediction accuracy and these two parameters when they set up the prototype to compress the datasets.

5.3.1 Compression ratio

Intuitively, if the compression ratio is higher, then more information about the data will be lost, thus leading to a

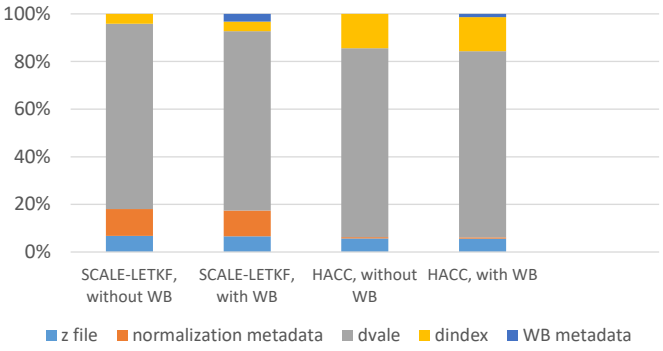


Fig. 11: The breakdown of the compressed data for datasets SCALE-LETKF and HACC when error bound is 0.001.

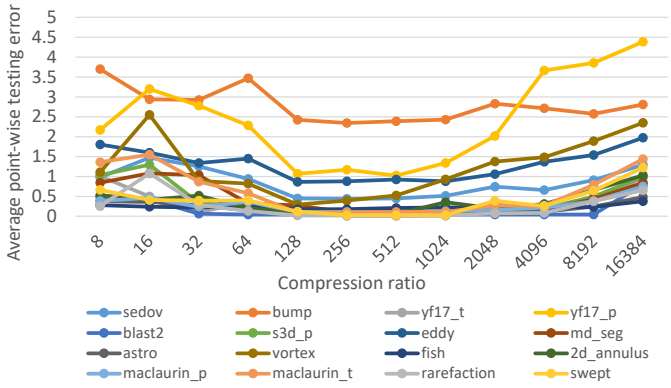


Fig. 12: The relationship between compression ratio and average point-wise relative testing error. The x-axis shows the theoretical compression ratio and the y-axis shows the average point-wise relative error for the testing process.

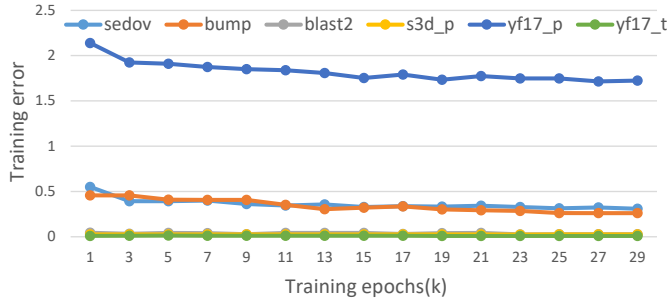


Fig. 13: The relationship between training epochs and training errors. The x-axis shows the number of epochs (each n in the y-axis means $n \cdot 1000$ epochs) and the y-axis shows the average training error.

higher prediction error. To verify this, we run experiments on all 16 small datasets. The results of these tests are shown in Figure 12, which indicates that, unlike traditional compressors, an autoencoder's compression ratio and prediction error do not have a positive correlation. The results also show that, when the theoretical compression ratio is between $128 \sim 1024$, the CAE can usually perform well.

Finding 5: ***Autoencoders, unlike conventional compressors, do not have a positive correlation between compression ratio and prediction error. In general, a theoretical compression ratio of*** $128 \sim 1024$ ***can result in a low prediction error.***

5.3.2 Training epochs

Intuitively, if we set a higher number of training epochs, then we will get better testing results, which means that the predictions will be more accurate. The drawback to this is that training a neural network can be extremely time consuming. If, however, the number of epochs is set very low, then the testing results will have very high prediction errors. To explore the relationship between the prediction error and the training epochs, we again run experiments on the six randomly chosen datasets, as shown in Figure 13. This figure shows that, at first, prediction errors decrease as the number of training epochs increases. After a certain point (normally around 15,000 epochs), however, the prediction errors should stabilize. To ensure that we have a low prediction error, all the experiments in this work are conducted with 25,000 training epochs unless otherwise mentioned.[8]

5.4 Training Time and Compression Throughput

Throughout the evaluation process, we notice that while the autoencoder can achieve very high compression ratios, it has a potential drawback: the lengthy training time. We record the training time when conducting previous tests, and the results are shown in Figures 14.

From Figure 14, we usually need at least one hour to train the autoencoder with 25,000 epochs. This can go up dramatically when a high compression ratio, such as 8,192 or 16,384, is desired. This is because the computation resource requirements of matrix calculation increase quadratically.

Conducting a training process for every dataset compression seems to be very time-consuming. However, we notice that for scientific data within the same application/benchmark, different variables (such as temperature, pressure, and velocity) share similar data features. This implies that the training time can be significantly reduced by reusing the knowledge from the other variables or timesteps of the same application/benchmark. We take advantage of this feature and use two new training methods to improve the training speed:

- *Reuse variable knowledge*: When compressing a variable, we reuse the training knowledge from another variable (from the same application/benchmark), which has already been trained. Assume an application generates N variables, then reusing variable knowledge can potentially improve the training speed by up to N times.
- *Reuse timesteps knowledge*: When compressing a variable, we only train part of the timesteps within that variable.

[8]When training the two large datasets mentioned in Section 5.2, we only use 10,000 epochs for training in order to reduce the training time. Figure 13 shows that, for most of the datasets, there is not too significant of a difference between the training errors with 10,000 and 25,000 epochs.

For the rest timesteps, we reuse the training knowledge from the previously trained timesteps. Assume the timesteps to train is $\frac{1}{T}$ of the total timesteps in the variable, then reusing timesteps knowledge can improve the training speed by T times.

In section 5.2.2, since the datasets (*SCALE-LETKF* and *HACC*) that we are testing have relatively large dataset sizes: 3.39GB and 1.69GB, when we conduct the training processes, we reuse the training knowledge from both variables and timesteps. In Figure 15, we record the training time with different combinations of N and T when we train the datasets *SCALE-LETKF* and *HACC*. The figure shows that, after reusing the training knowledge, the training time is reduced to around $\frac{1}{1200}$ of the original total training time for *SCALE-LETKF* when we choose (N, T) to be $(12, 100)$; and the training time is reduced to around $\frac{1}{300}$ of the original total training time for *HACC* when we choose (N, T) to be $(3, 100)$. For the final training of the two datasets, when reusing timestep knowledge, for both of the datasets, we use 1% of the total timesteps of the same variable; when reusing variable knowledge, we only train one variable and reuse the training knowledge for other variables within the dataset (*SCALE-LETKF* and *HACC* have 12 variables and three variables, respectively). The compression ratio results shown in Figure 10 indicates that after using the new training methods, our compression autoencoder still has a 2 to 4X compression ratio gain over SZ, and a 10 to 50X compression ratio gain over ZFP. In our future work, we will conduct further research on how different variables and number of T would influence the prediction accuracy and compression ratio of the compression autoencoder; we will also investigate how to choose the appropriate variable and number of T to achieve an optimal tradeoff between the training time and compression ratio.

Finding 6: ***The compression autoencoder's training overhead can be significantly alleviated by reusing the training knowledge from both variables and timesteps in the same application/benchmark.***

When compressing the datasets, our autoencoder has an average compression throughput of around 30 MB/s. This throughput is less than SZ or ZFP, which can achieve a compression throughput of around 60 MB/s on the same file. Table 3 shows the comparison of compression throughput and compression ratio of datasets *SCALE-LETKF* and *HACC* under relative error bound 0.1. From the table, SZ has a compression throughput gain over CAE for 1.72 and 1.69, but CAE has a compression ratio gain over SZ for 2.21 and 5.81 for the two datasets, respectively; Likewise, ZFP has a compression throughput gain over CAE for 4.08 and 1.33, but CAE has a compression ratio gain over ZFP for 32.62 and 115.63 for the two datasets, respectively. In conclusion, the autoencoder can achieve a higher compression ratio than SZ or ZFP for more than up to 100 times, while SZ and ZFP have a compression throughput gain over CAE for up to 4 times. Also, it is worth noting that all the experiments in this paper are conducted on a single machine without GPU support, as the training and testing performance of the machine

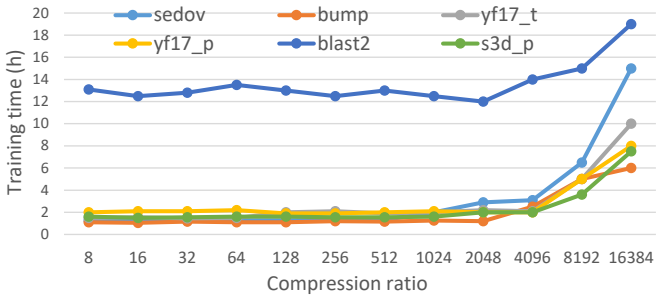


Fig. 14: Training time. The x-axis shows the theoretical compression ratio and the y-axis shows the training time in hours.

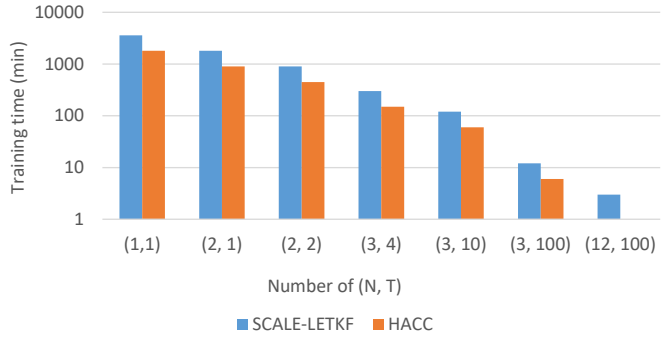


Fig. 15: The training time for datasets *SCALE-LETKF* and *HACC*. The x-axis shows the different combinations of N and T when reusing variable knowledge and timestep knowledge for training. N means one variable's training knowledge is used for N variables; T means only $\frac{1}{T}$ of the total timesteps are used for training.

learning process is not our focus in this paper. However, we can provide several possible solutions to address this issue further.

The future trend [41], [42], [43] indicates that while improving storage space is hard to achieve, there remain lots of potentials to improve computational power, especially with the upsurge of supercomputers and hardware acceleration (such as FPGA, GPU, and TPU) [44], [45]. For example, Heeswijk et al. [46] and Raina et al. [47] find that, depending on the application, speedups of up to 300 times are possible by executing code on a single GPU instead of a typical CPU; it is possible to obtain even higher speedups by using multiple GPUs. Another typical way to accelerate the machine learning training process is by utilizing the parallelism of supercomputers/multi-machines, related work [48], [49] indicates that this technique can provide a speedup of up to 450.65x. A further possible solution that has been proposed to alleviate the training overhead is transfer learning [50], which has the potential to vastly reduce training times. In addition, recent research [51] indicates that computer speeds are increasing much faster than storage technology capacities and I/O rates.

As a result, we believe that the issue of relatively low compression throughput for autoencoder is addressable and that the compression autoencoder is promising for high-ratio scientific data reduction.

TABLE 3: The comparison of compression throughput and compression ratio of datasets *SCALE-LETKF* and *HACC*. CAE/SZ and CAE/ZFP show the ratio of CAE's performance over SZ and ZFP for each corresponding metric in the table.

Metrics	Datasets	Raw Performance			Ratio	
		CAE	SZ	ZFP	CAE/SZ	CAE/ZFP
Compression throughput	SCALE-LETKF	15.6 (MB/s)	26.95 (MB/s)	63.71 (MB/s)	0.58	0.24
	HACC	55.39 (MB/s)	92.7 (MB/s)	73.33 (MB/s)	0.60	0.76
Compression ratio	SCALE-LETKF	72.10	32.54	2.21	2.22	32.62
	HACC	241.67	41.57	2.09	5.81	115.63

6 Related Work

In order to improve the compression ratio of HPC scientific data, many lossy data compressors have been proposed in recent years. SZ [27] aims to accurately approximate the original data by deploying multiple curve-fitting models to encode data streams. Another common HPC lossy compressor that also involves curve fitting is ISABELA [25]. ISABELA converts the multi-dimensional floating-point arrays in snapshots to sorted data-series before performing data compression by B-spline interpolation. Unlike SZ and ISABELA, ZFP [26] is motivated by fixed-rate encoding and random access. ZFP follows the classic texture compression for image data and involves fixed-point integer conversion, block transform bit-plane encoding, and more.

As a type of artificial neural network typically used for dimensionality reduction, autoencoders have the potential to address an increasing need for flexible lossy compression algorithms. Recently, researchers in various fields have applied autoencoders for compression. In the image compression field, Theis et al. [30] aim at directly optimizing the rate-distortion tradeoff produced by an autoencoder. Romero et al. [32] apply the autoencoder to compress quantum data. Testa et al. [33] attempt to deploy autoencoders to compress biosignals generated by IoT devices at low computational costs while also providing a high compression ratio. To the best of our knowledge, this paper is the first work to explore the autoencoder as a lossy compressor for scientific data compression.

7 Conclusion and Future Work

In this paper, we conduct comprehensive evaluations on using an autoencoder as a lossy compressor on HPC datasets. We perform experiments on 18 real-world HPC scientific datasets and show that simply using an autoencoder to compress the datasets likely leads to very large prediction errors. However, we explain that this can be largely solved by the tuning schemes we propose regarding the prediction accuracy and storage overhead. After our tuning operations, the autoencoder can achieve much higher compression ratios than those of the state-of-art HPC lossy compressors SZ (2 to 4X) and ZFP (10 to 50X) under common error-bounds. In addition, we present users with guidance on how to quickly determine whether a dataset is suitable for compression via an autoencoder. We also show the relationships between several important metrics (namely compression ratios, training epochs, and prediction errors) that can be used to guide users to configure the autoencoder to their preferences when training the autoencoder and compressing files. We also present two remaining challenges for using an autoencoder as a lossy compressor: the relatively loose error-bound as well as the relatively low compression throughput. In the future, we plan to explore lossy data compression with the autoencoder under stricter error bounds and attempt to use hardware acceleration techniques, such as GPU parallel computing, to improve computation performance.

The source codes for our autoencoder prototype as well as the scientific datasets we used are publicly available at https://github.com/tobivcu/autoencoder.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and feedback. This work was partially supported by the US National Science Foundation NSF-1828363, NSF-1813081, CCF-1718297, CCF-1812861, and NJIT research startup fund. The authors also wish to acknowledge the support from the NSF Chameleon cloud which is used for some of the experiments.

References

[1] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[2] "Zip (file format) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Zip_(file_format), [Online; accessed 8-July-2018].

[3] "GNU gzip," https://www.gnu.org/software/gzip/, [Online; accessed 8-July-2018].

[4] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, 2009.

[5] "LZ4, extremely fast compression." https://github.com/lz4/lz4.

[6] S. Mittal and J. S. Vetter, "A survey of architectural approaches for data compression in cache and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[7] L. Whitehouse, "Understanding data deduplication ratios in backup systems," https://searchdatabackup.techtarget.com/tip/Understanding-data-deduplication-ratios-in-backup-systems, 2015.

[8] G. J. Sullivan, J.-R. Ohm, W.-J. Han, T. Wiegand *et al.*, "Overview of the high efficiency video coding(hevc) standard," *IEEE Transactions on circuits and systems for video technology*, 2012.

[9] U. Hafner, "Fiasco-an open-source fractal image and sequence codec," *Linux Journal*, vol. 2001, no. 81es, p. 3, 2001.

[10] G. K. Wallace, "The jpeg still picture compression standard," *Communications of the ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[11] D. Le Gall, "Mpeg: A video compression standard for multimedia applications," *Communications of the ACM*, 1991.

[12] K. Brandenburg, "Mp3 and aac explained," in *Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding*. Audio Engineering Society, 1999.

[13] T. Lu, E. Suchyta, D. Pugmire, J. Choi, S. Klasky, Q. Liu, N. Podhorszki, M. Ainsworth, and M. Wolf, "Canopus: A paradigm shift towards elastic extreme-scale data analytics on hpc storage," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 58–69.

[14] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*.

[15] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, 2016.

[16] J. Zhang, X. Zhuo, A. Moon, H. Liu, and S. W. Son, "Efficient encoding and reconstruction of hpc datasets for checkpoint/restart," in *IEEE... Symposium on Mass Storage Systems and Technologies*, 2019.

[17] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 7.

[18] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C.-S. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross *et al.*, "Isobar preconditioner for effective and high-throughput lossless data compression," in *2012 IEEE 28th international conference on data engineering*.

[19] N. Shah, E. R. Schendel, S. Lakshminarasimhan, S. V. Pendse, T. Rogers, and N. F. Samatova, "Improving i/o throughput with primacy: preconditioning id-mapper for compressing incompressibility," in *2012 IEEE international conference on cluster computing*. IEEE, 2012, pp. 209–219.

[20] J. Wang, T. Liu, Q. Liu, X. He, H. Luo, and W. He, "Compression ratio modeling and estimation across error bounds for lossy compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 7, pp. 1621–1635, 2019.

[21] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Fixed-psnr lossy compression for scientific data," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018.

[22] X. Zou, T. Lu, W. Xia, X. Wang, W. Zhang, H. Zhang, S. Di, D. Tao, and F. Cappello, "Performance optimization for relative-error-bounded lossy compression on scientific data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 7, 2020.

[23] H. Luo, D. Huang, Q. Liu, Z. Qiao, H. Jiang, J. Bi, H. Yuan, M. Zhou, J. Wang, and Z. Qin, "Identifying latent reduced models to precondition lossy compression," in *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'19)*, 2019.

[24] Y. Jin, S. Lakshminarasimhan, N. Shah, Z. Gong, C.-S. Chang, J. Chen, S. Ethier, H. Kolla, S.-H. Ku, S. Klasky *et al.*, "S-preconditioner for multi-fold data reduction with guaranteed user-controlled accuracy," in *2011 IEEE 11th International Conference on Data Mining*. IEEE, 2011, pp. 290–299.

[25] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," in *European Conference on Parallel Processing*. Springer, 2011, pp. 366–379.

[26] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[27] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 730–739.

[28] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, and T. Liu, "Understanding and modeling lossy compression schemes on hpc scientific data," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[29] H. Kamyshanska and R. Memisevic, "The potential energy of an autoencoder," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 6, pp. 1261–1273, 2014.

[30] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy image compression with compressive autoencoders," *arXiv preprint arXiv:1703.00395*, 2017.

[31] A. Sento, "Image compression with auto-encoder algorithm using deep neural network (dnn)," in *Management and Innovation Technology International Conference (MITicon), 2016*. IEEE, 2016.

[32] J. Romero, J. P. Olson, and A. Aspuru-Guzik, "Quantum autoencoders for efficient compression of quantum data," *Quantum Science and Technology*, vol. 2, no. 4, p. 045001, 2017.

[33] D. Del Testa and M. Rossi, "Lightweight lossy compression of biometric patterns via denoising autoencoders," *IEEE Signal Processing Letters*, vol. 22, no. 12, pp. 2304–2308, 2015.

[34] "Tensorflow, an open source machine learning framework for everyone." https://www.tensorflow.org/.

[35] "Google Colaboratory Environment," https://colab.research.google.com/notebooks/welcome.ipynb.

[36] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM computing surveys (CSUR)*, vol. 27, no. 3, 1995.

[37] "bzip2, a freely available, high-quality data compressor." https://web.archive.org/web/19980704181204/http://www.muraroa.demon.co.uk/.

[38] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An efficient transformation scheme for lossy data compression with point-wise relative error bound," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 179–189.

[39] X. Zou, T. Lu, S. Di, D. Tao, W. Xia, X. Wang, W. Zhang, and Q. Liao, "Accelerating lossy compression on hpc datasets via partitioning computation for parallel processing," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 1791–1797.

[40] S. Di and F. Cappello, "Optimization of error-bounded lossy compression for hard-to-compress hpc data," *IEEE transactions on parallel and distributed systems*, vol. 29, no. 1, pp. 129–143, 2017.

[41] S. Binkley, "Future trends in advanced scientific computing for open science," https://science.energy.gov/~/media/hep/hepap/pdf/20150406/day2/2015-0407-HEPAP-future-directions-01Binkley.pdf.

[42] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money, "Big data: Issues and challenges moving forward," in *System sciences (HICSS), 2013 46th Hawaii international conference on*. IEEE, 2013.

[43] M. Padgavankar and S. Gupta, "Big data storage and challenges," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 2, 2014.

[44] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. Ahrens, "Understanding gpu-based lossy compression for extreme-scale cosmological simulations," in *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'20)*. IEEE, 2020.

[45] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "wavesz: a hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 74–88.

[46] M. Van Heeswijk, Y. Miche, E. Oja, and A. Lendasse, "Gpu-accelerated and parallelized elm ensembles for large-scale regression," *Neurocomputing*, vol. 74, no. 16, pp. 2430–2437, 2011.

[47] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009.

[48] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014.

[49] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng, "Map-reduce for machine learning on multicore," in *Advances in neural information processing systems*, 2007, pp. 281–288.

[50] T. Liu, S. Alibhai, J. Wang, Q. Liu, X. He, and C. Wu, "Exploring transfer learning to reduce training overhead of hpc data in machine learning," in *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2019, pp. 1–7.

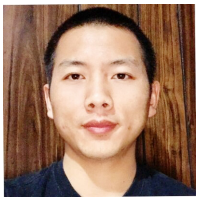
[51] I. Foster, M. Ainsworth, B. Allen, J. Bessac, F. Cappello, J. Y. Choi, E. Constantinescu, P. E. Davis, S. Di, and W. Di, "Computing just what you need: online data analysis and reduction at extreme scales," in *European Conference on Parallel Processing*. Springer, 2017, pp. 3–19.

Tong Liu received the B.S. degree in computer science from Huazhong University of Science and Technology, China, in 2015. He is currently a fifth-year PhD student at Temple University. His research interests include computer systems, data storage, cloud computing, high performance computing, and data reliability.

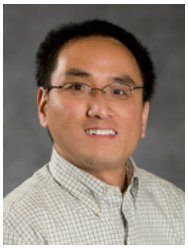
Jinzhen Wang is currently a third-year PhD student in the Department of Electrical and Computer Engineering at NJIT. He received his B.S. from Shandong University, China, in 2015 and his M.S. in Electrical Engineering from NJIT in 2017. His research interests include High Performance Comptuting and Cloud Computing.

Qing Liu is an Assistant Professor in the Department of Electrical and Computer Engineering at NJIT and Joint Faculty with Oak Ridge National Laboratory. Prior to that, he was a staff scientist at Computer Science and Mathematics Division, Oak Ridge National Laboratory for 7 years. He received his Ph.D. in Computer Engineering from the University of New Mexico in 2008, M.S. and B.S., from Nanjing University of Posts and Telecom, China, in 2004 and 2001, respectively.

Shakeel Alibhai received the B.S. degree in computer science from the Computer and Information Science department of Temple University in 2020. In addition to computer science, he minored in mathematics and completed a certificate in data science. His research interests include data storage, high-performance computing, and machine learning.



Tao Lu received the B.S. and M.S. degrees in Computer Science from Huazhong University of Science and Technology, China, in 2009 and 2012, respectively, and the Ph.D. degree in Electrical and Computer Engineering from Virginia Commonwealth University, in 2016. He is currently a senior software engineer in Marvell Semiconductor Inc. His research interests include computer systems, virtualization and cloud computing, high performance computing, and computer system security.



Xubin He received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1997, respectively, and the PhD degree in electrical engineering from University of Rhode Island, Kingston, RI, in 2002. He is currently a professor in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA. His research interests include computer architecture, data storage systems, virtualization, and high availability computing. Dr. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the Sigma Xi Research Award (TTU Chapter) in 2005 and 2010. He is a senior member of the IEEE, a member of the IEEE Computer Society and USENIX.