Reducing the Training Overhead of the HPC Compression Autoencoder via Dataset Proportioning

Tong Liu[1], Shakeel Alibhai[1], Jinzhen Wang[2], Qing Liu[2], and Xubin He[1]

[1]*Department of Computer and Information Science, Temple University, Philadelphia, PA*
[2]*Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ*

***Abstract*—As the storage overhead of high-performance computing (HPC) data reaches into the petabyte or even exabyte scale, it could be useful to find new methods of compressing such data. The compression autoencoder (CAE) has recently been proposed to compress HPC data with a very high compression ratio; however, this machine learning-based method suffers from the major drawback of lengthy training times. In this paper, we attempt to mitigate this problem by proposing a proportioning scheme that reduces the amount of data that is used for training relative to the amount of data to be compressed. We show that this method drastically reduces the training time without, in most cases, significantly increasing the error. We further explain how this scheme can even improve the accuracy of the CAE on certain datasets. Finally, we provide some guidance on how to determine a suitable proportion of the training dataset to use in order to train the CAE for a given dataset.**

***Index Terms*—Data compression, HPC, machine learning, autoencoder, training time**

I. Introduction

The amount of data produced by high-performance computing (HPC) applications is very significant: it is often reported to be on the terabyte or petabyte scale [1], and is growing close to—if not already at—the exabyte scale [2], [3]. With such a massive quantity of data, it is extremely important to have an efficient way to compress it; otherwise, the storage overhead of the data could be prohibitively large.

In general, there are two classes of compression methods: lossless and lossy compressors. Lossless compressors are able to compress data such that the compressed file can be decompressed without losing any data. However, the tradeoff of lossless compressors is that their compression ratios are relatively low. Lossy compressors, conversely, are able to achieve much higher compression ratios, with the drawback that some data loss generally occurs when decompressing the data. Several lossy compressors have been proposed for compressing HPC data, with some of the most widely-known including SZ [18], ZFP [17], and ISABELA [4], [16].

Another method of compressing HPC data that has recently been proposed utilizes machine learning. Specifically, the technique uses a type of neural network known as an autoencoder. A special autoencoder, known as the compression autoencoder (CAE), has been developed for compressing HPC data; it has been indicated that it can achieve a compression ratio of up to two orders of magnitude on certain datasets [5]. Such a high compression ratio makes it very appealing to use. However, one of its main drawbacks is that the training process of the CAE can take a very long time. This paper proposes a way to mitigate this problem by proportioning the amount of training data to use relative to the amount of data to be compressed.

The rest of the paper is organized as follows. Section 2 discusses autoencoders in more detail, giving special attention to the CAE. Section 3 proposes our solution for reducing the lengthy training times of the CAE and explains our experimental testbed. Section 4 describes the results, specifically focusing on the impact on training time and testing error. Section 5 offers guidance on determining a suitable training dataset size relative to the size of the compression dataset. Section 6 discusses related work. The conclusion is presented in Section 7.

II. Background and Motivation

The basic idea of an autoencoder, a type of unsupervised neural network, is that it takes some input and produces an output with as little loss of data as possible [6]. Autoencoders generally have two main parts, an encoder and a decoder. The encoder may contain a number of layers of data points. In many cases, the amount of data points would decrease from one layer to the next, thereby accomplishing dimensional reduction. Following the encoder is the decoder, which contains the same number of layers that the encoder has. However, whenever the number of data points decreases from one layer to the next in the encoder, the number of data points would increase between the two corresponding layers in the decoder.

In between the encoder and the decoder is the middle layer, sometimes known as the Z layer. The Z layer contains the minimal number of data points of all the layers.

The CAE [5] is an autoencoder specially designed for compressing HPC data. In the CAE, the compressor is represented by the encoder portion of the autoencoder, which consists of three layers of decreasing dimensions. For example, the first layer may contain n nodes, the second layer may contain $n/8$ nodes, and the third layer may contain $n/64$ nodes, where n is the number of data points sent to the autoencoder at a time. In this case, every data point in the original input file is represented by one node in the first layer of the autoencoder; by the second layer, however, one node represents eight data points. This increases to every 64 data points being represented per node for the third layer. Thus, in every layer, the number of data points needed to store the representation is reduced by

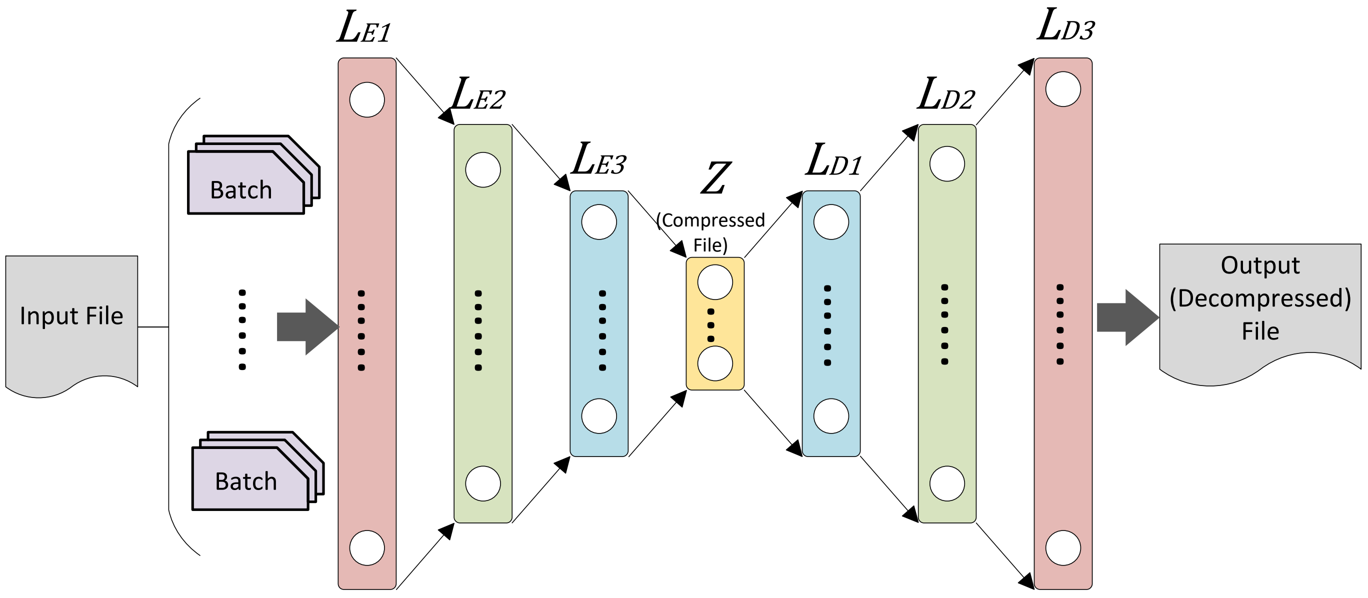

Fig. 1. Diagram of a seven-layer CAE with three layers in the encoder, a Z layer, and three layers in the decoder. Note that the entirety of the input file does not always proceed through the autoencoder at once, but may be split into multiple batches first.

$8x$, meaning that the theoretical compression ratio increases $8x$ with each layer.

After the encoder is the Z layer (the middle layer), which may contain $n/512$ nodes (another $8x$ reduction). In this layer, which represents the compressed file, each node represents 512 data points of the original file; the theoretical compression ratio would thus be 512.

Following the Z layer is the decompressor (represented by the decoder portion of the autoencoder), which contains three layers of increasing dimensions. The last layer of the decoder would contain n data points, and this layer represents the decompressed file. The CAE is thus a seven-layer autoencoder, as there are three layers in the encoder, one middle layer, and three layers in the decompressor.

In order to move from one layer to the next, a series of weight matrices and bias vectors is used. There is one weight matrix and one bias vector between every two layers of the autoencoder; therefore, a seven-layer autoencoder would have six weight matrices and six bias vectors. These weight matrices and bias vectors are generally obtained through a training process. Before compressing a dataset D, a dataset D' (which is generally similar to dataset D) will go through the autoencoder for multiple epochs. In each epoch, the autoencoder will compress and decompress D'—also known as the "training dataset"—by using the weight matrices and bias vectors (which were initialized prior to the start of the first epoch[1]), and then compare the output of the decompressed D' to the original data points in D'. It will then try to minimize that difference by modifying the weight matrices and bias vectors. It does this every epoch. At the end of the training phase, the weight matrices and bias vectors will be saved; they can then be used to compress dataset D. In this work, as in previous work [5], the compression process is often referred to as the "testing" process, and the dataset to be compressed (e.g. D) is also referred to as the "testing" dataset.

One of the major drawbacks facing the compression autoencoder is that its training time can be very lengthy. Previous work [5] found that, with compression ratios between 64 and 512, training a dataset with less than 50,000 data points over 25,000 epochs could take around two hours. If the training dataset contained around 300,000 data points, then the training time could increase to over 12 hours. Such findings establish a correlation between the size of the training dataset and the amount of time required for training. As the training and testing datasets are equal in size in these experiments, if the dataset for compression (i.e. the testing dataset) was very large, then the training dataset would also be very large and could take a significant amount of time to train. Therefore, in this work we focus on the training overhead problem and propose a proportioning method to address the issue.

III. Design

A. Method

As mentioned previously, before a dataset D is compressed, weights and biases are obtained by training a dataset D', which is generally similar to D. Dataset D' would often be generated from the same HPC scientific benchmark as dataset D, and could even be a portion of dataset D.

Since datasets D and D' are from the same benchmark, they often have similar features; thus, the weights and biases of one would likely be applicable to the other as well. As the features of both datasets are generally similar, we propose that

[1]The autoencoder model used in previous work [5] and in these experiments initialized the weight matrices to values from the random normal distribution and the bias vectors to zero. Different autoencoder models could potentially have different initialization methodologies.

TABLE I
OVERVIEW OF THE 18 EXPERIMENTAL DATASETS

Dataset	Data Points	Benchmark
Bump	25,000	NEK5000
Astro	32,768	NEK5000
Fish	32,768	NEK5000
Sedov-Pres	39,072	FLASH
Sedov-Temp	39,072	FLASH
yf17Pres	48,552	NEK5000
yf17Temp	48,552	NEK5000
Swept	77,180	NEK5000
S3DP	97,020	MCERI
Inviscid Vortex	100,000	NEK5000
Rarefaction	100,000	NEK5000
Maclaurin-Pres	133,376	FLASH
Maclaurin-Temp	133,376	FLASH
Eddy	135,000	NEK5000
2DAnnulus	181,890	NEK5000
Blast2	289,440	FLASH
MD-Seg	300,000	GROMACS
mhdTime	2,400,001	SDR

by reducing the size of the training dataset (without changing the size of the compression, or testing, dataset), we could reduce the training time while still learning enough features to maintain an acceptable level of accuracy. In the previously used CAE [5], the size of the training and testing HPC datasets would be the same. Assume that X represents the number of data points in each of these datasets. However, it may not be necessary to train all X data points from the training dataset in order to generate weights and biases for the testing dataset; rather, the features could be learned from the training dataset by using a portion of X. We thus develop a method known as **dataset proportioning**: finding a rough proportion of the training dataset to use for compression that can reduce the training time while still learning enough features from the training dataset to be able to compress and decompress the testing dataset without losing a significant amount of accuracy.

B. Experimental Setup

To test our theory, we set up an experimental testbed. The experimental setup is the same seven-layer compression autoencoder used in the previous work [5]. As displayed in Figure 1, it has three layers of decreasing dimensions for the encoder (representing the compressor), a middle layer (representing the compressed file), and three layers of increasing dimensions for the decoder (representing the decompressor). The main difference between this autoencoder and the previous CAE is that this autoencoder only uses a portion of the training dataset for training. In our experimental model, the n percent of each training dataset that is used is the first n percent. For example, when training is conducted on 25% of the training dataset, it is the first 25% of data points that are chosen for training. Future work may look into whether choosing a different section of the dataset, or whether, for example, choosing one in every four data points, may yield better results.

We use this autoencoder to run experiments on the 18 datasets listed in Table I. These datasets come from HPC simulations from several open-source benchmarks: NEK5000 [11], FLASH [12], MCERI [13], GROMACS [14], and SDR [15]. Each of these datasets has two parts, one for training and one for testing. (For example, the two parts of the *MD-Seg* dataset are md_training.bin and md_testing.bin.) The "Data Points" column of the table indicates the number of double-precision floating-point data points in each of these parts, as the training and testing portions are equal in size.

For each of the 18 datasets, training is generally performed six times: once on the entire training dataset, as well as on approximately 50, 25, 10, 5, and 1 percent of the training dataset. Each training is performed over 10,000 epochs. After each training finishes, weights and biases from the training process are saved. These weights and biases are then used to compress the entire corresponding testing dataset. For example, for the *MD-Seg* dataset, training is performed using the six aforementioned proportions of the training dataset, md_training.bin, and then is tested on the testing dataset, md_testing.bin, with each of the six pairs of weights and biases.

IV. RESULTS AND EVALUATION

A. Training Time

The motivation for this work is to attempt to reduce the lengthy training times encountered when using the autoencoder for lossy compression of HPC data. Figure 2 displays the training times for 17 of the experimental datasets as the size of each training dataset decreases.[2]As can be seen from this graph, as the size of the training dataset is reduced, the time required for training is generally reduced as well. While this is not always as obvious for smaller datasets, which already have a relatively low training time, the differences are much more pronounced for large datasets. Several of the large datasets experience a training time reduction of an order of magnitude. It appears that these datasets' training times approach a horizontal asymptote at around 40 seconds; however, the exact value of the training time limit may depend on the specific autoencoder model and the hardware capability.

B. Testing Error

From Figure 2, it is clear that using only a portion of the training dataset for the training process can significantly reduce the training time. However, lowering the training time would be of little use if doing so results in high prediction errors. Therefore, it is important to examine the errors to ensure that they are still acceptable.

The testing prediction errors are presented in Figure 3. From the graph, we can divide the datasets into three main categories, as indicated in Table II. The first category consists

[2]The training time of the mhdTime dataset is not included in this figure because, as its size is very large relative to the experimental datasets, it has a very large training time and thus skews the proportions of the graph, making the trend lines for the other datasets harder to see. The mhdTime dataset does, however, follow the same pattern observed with the other datasets.

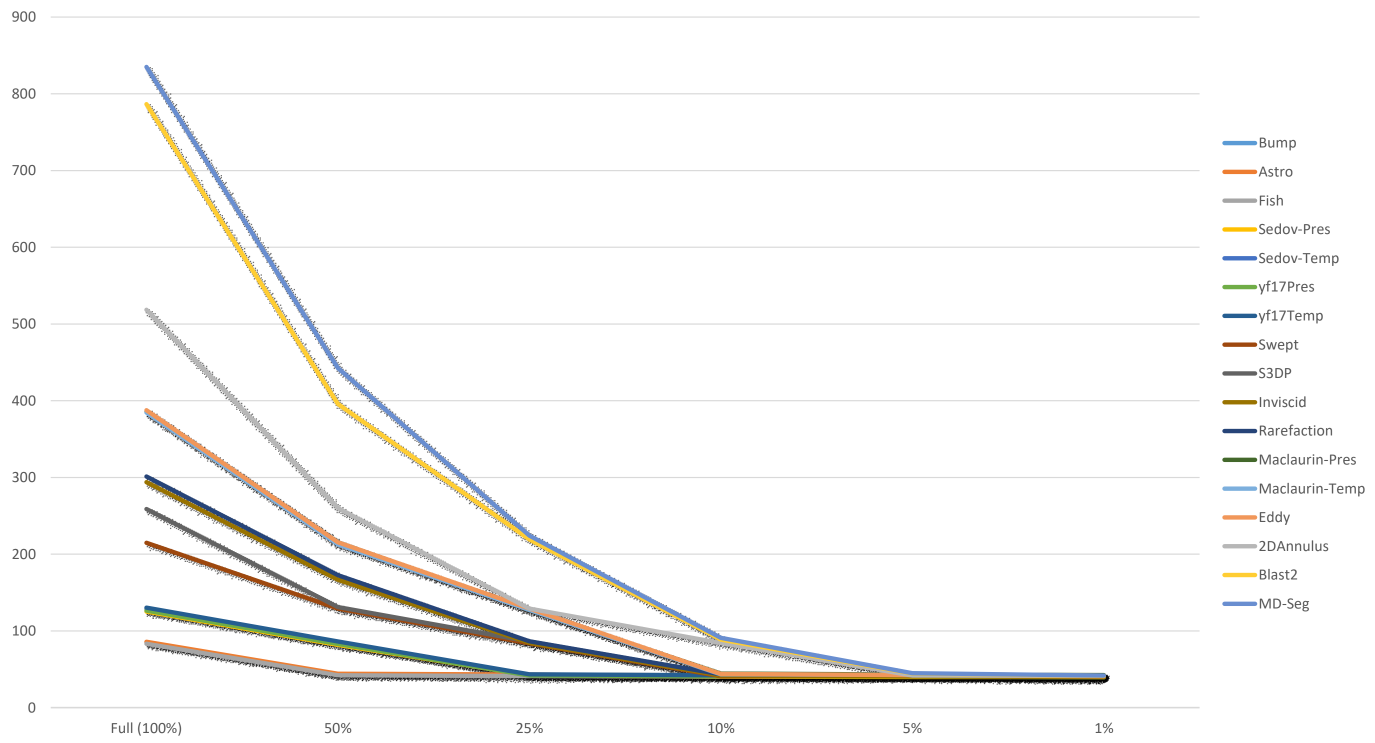

Fig. 2. Training times as the size of the training dataset decreases. The x-axis represents the size of the training dataset used relative to its original size (and, since the testing dataset would have an equal original size, relative to the testing dataset). The y-axis represents the number of seconds required for training.

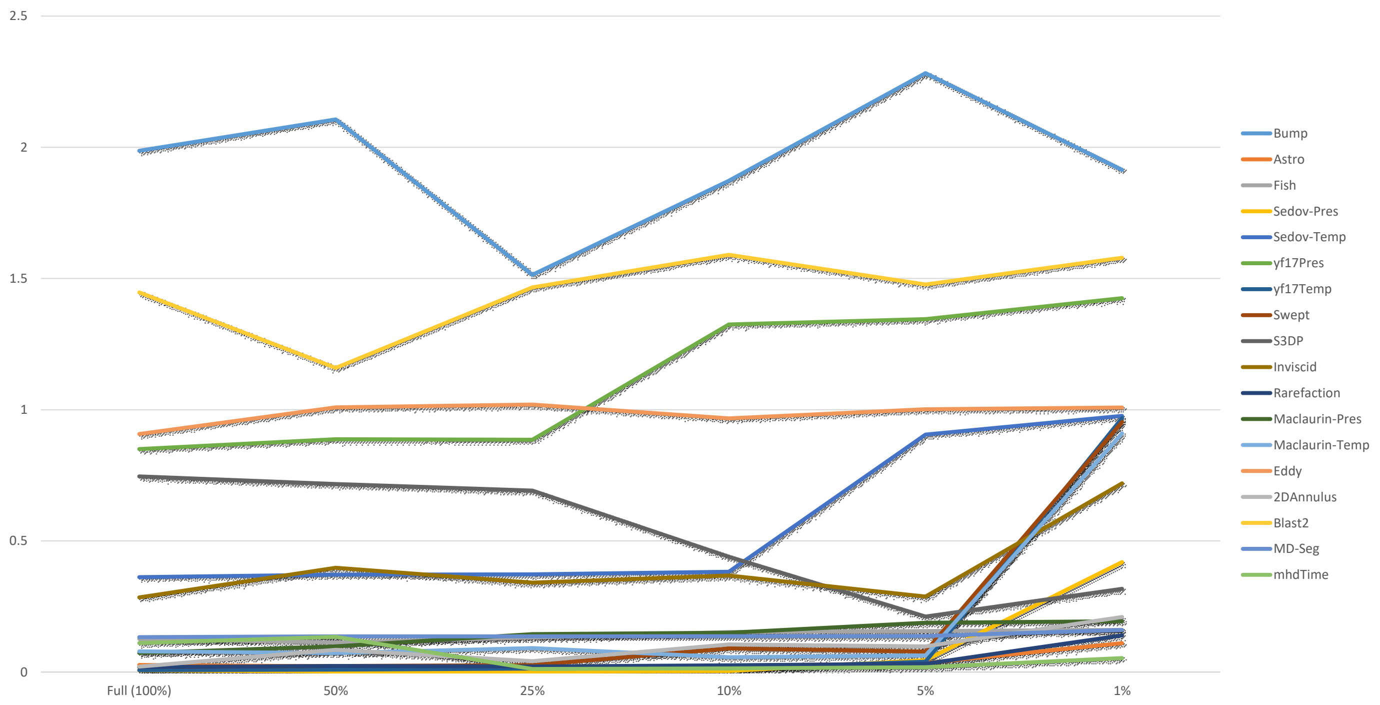

Fig. 3. Testing prediction errors on the 18 experimental datasets. The x-axis represents the size of the training dataset that was used relative to its original size. The y-axis represents the prediction error after weights and biases from training a certain size of the training dataset were used to compress the entire testing dataset. Note that, for example, “1” in the y-axis represents 100% error.

TABLE II
EXPERIMENTAL DATASETS ORGANIZED BY TESTING PREDICTION PERFORMANCE

Dataset	Category	CV (Testing Dataset)	CV (Training Dataset)
Astro	Good	0.067231	0.060980
Fish	Good	3.547695	3.590917
Sedov-Pres	Good	0.000473	0.000473
yf17Temp	Good	0.007613	0.015392
Swept	Good	0.127470	0.042109
Rarefaction	Good	0.043189	0.044695
Maclaurin-Pres	Good	0.174820	0.175301
Maclaurin-Temp	Good	0.629824	0.631273
2DAnnulus	Good	0.208714	0.175425
MD-Seg	Good	0.221210	0.220753
mhdTime	Good	0.594166	0.482194
Blast2	Bad	0.639798	0.639526
Bump	Bad	0.641916	0.239242
Sedov-Temp	Bad	0.428897	0.428632
yf17Pres	Bad	1.381044	0.919403
Inviscid	Bad	0.715134	0.509440
Eddy	Bad	0.757119	0.658896
S3DP	Special Case	0.037480	0.790152

of datasets that have high prediction errors on the testing dataset, regardless of whether the full training dataset is used for training or whether only a portion of it is used. The second category includes datasets that have low prediction errors when the full training dataset is used, as well as frequently have low errors when a portion of the training dataset is used. The third category, which only contains a single dataset, indicates a dataset that has a high prediction error when the full training dataset is used, but its prediction error decreases significantly when only a portion of it is used.

The first category (high prediction errors regardless of the size of the training dataset) consists of the datasets *Bump*, *Sedov-Temp*, *yf17Pres*, *Inviscid*, *Eddy*, and *Blast2*. 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. As can be verified in Table II, all of these datasets have high CV values. This validates the theory from the previous work [5] that the CV can be a good indicator of whether the autoencoder is well-suited for compressing a given dataset.

The second category (low prediction errors on the full training dataset as well as often low prediction errors on a portion of the training dataset) contains the datasets *Astro*, *Fish*, *Sedov-Pres*, *yf17Temp*, *Swept*, *Rarefaction*, *Maclaurin-Pres*, *Maclaurin-Temp*, *2DAnnulus*, *MD-Seg*, and *mhdTime*. With one exception, all of these datasets have low CV values, again validating the previous theory. The one exception to this is the *Fish* dataset: it has a very high CV of over 3.5, yet it still performs well with the autoencoder. This is likely because this dataset contains a large number of zero values.

One dataset that appears to be an outlier to both of these groups is the *S3DP* dataset. When training was performed on the full training dataset and those weights and biases were then used to compress the testing dataset, the testing prediction error was very high—about 74.6%. However, when smaller amounts of the training dataset were used to generate weights and biases, the testing error gradually decreased until it reached a prediction error around only 21.2%; that error was reached when 5% of the training dataset was used for training. This is likely because the training and testing datasets of *S3DP* are very different: as mentioned in Table II, the training part of *S3DP* has a CV of around 0.79, but the testing part has a CV of under 0.04. It is also noteworthy that *S3DP* is the only experimental dataset in these experiments to have such a vast difference of CVs between the training and testing parts. From this, we can conclude that reducing the amount of the training dataset used to generate weights and biases can not only drastically reduce the training time without significantly increasing the error, but it could also improve the prediction accuracy for datasets whose training and testing portions have very different CV values.

V. Guidance on Training Dataset Proportion

A question that may arise here is how much of the training dataset we should use in order to generate weights and biases. Since we can assume that the "full" training dataset is equal to the size of the testing dataset, we could consider, for example, 50% of the training dataset to be essentially the same as 50% of the testing dataset.

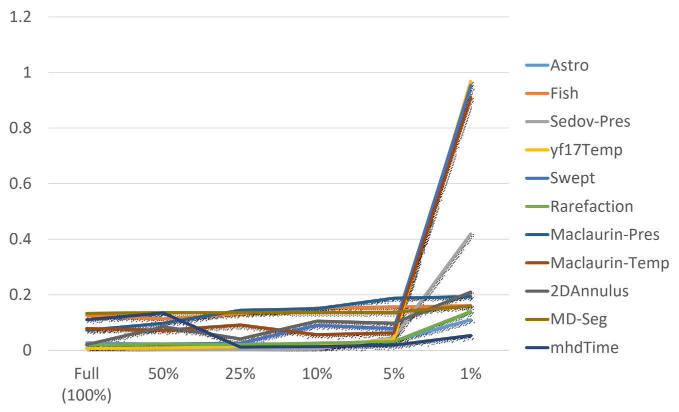

Fig. 4. Testing prediction errors on only the datasets that work well with the autoencoder. This graph contains the same data as Figure 3 except that the datasets with bad prediction errors have been removed.

Figure 4 displays the testing prediction errors of datasets that work well with the autoencoder. From this graph, it appears that almost all of the datasets which work well with the autoencoder still have difficulty performing well when only 1% of the training dataset is used; thus, we can eliminate this proportion from our examination.

Figure 5 shows the prediction errors of the "good" datasets as the size of the training dataset decreases from 100% to 5% of the size of the testing dataset. While some of these datasets may appear to have drastic shifts, it is important to remember that the compression autoencoder prototype is error-bounded;

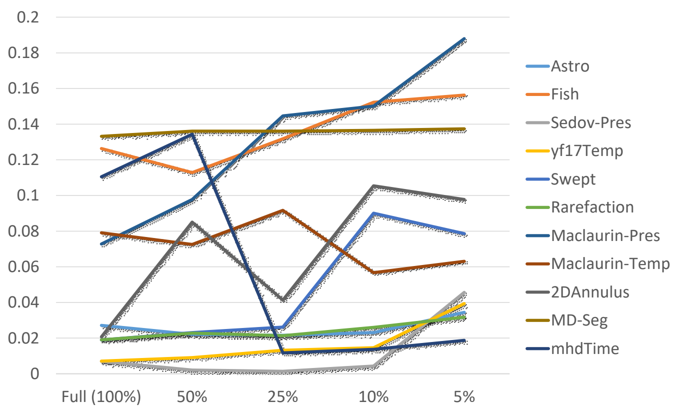

Fig. 5. Testing prediction errors on only the datasets that work well with the autoencoder, going down to 5%. This graph is essentially a "zoomed in" version of Figure 4.

therefore, a jump from a mean error of, for example, 2% to 6% may not significantly impact the accuracy if the error-bound is held constant. (The inclusion of an error-bound also means that a jump in error could decrease the compression ratio; however, this could then indicate a trade-off between training speed, accuracy, and compression ratio.) Most of these "good" datasets appear to hold well all the way down to 10%, and most of them do not significantly increase their error for 5% either. However, there are two datasets whose mean testing errors do significantly increase when the size of the training dataset is reduced from 10% to 5%: *Sedov-Pre*s and *yf17Temp*. The testing error of *Sedov-Pres* increases by around $10x$ when the training dataset's proportion is reduced from 10% to 5%, while the testing error of *yf17Temp* increases by almost $3x$. This likely also has to do with CV values: At 0.000473, *Sedov-Pres* has the lowest CV of the experimental datasets; *yf17Temp* has the second lowest at just 0.007613. None of the other experimental datasets have a CV that is even in the same magnitude as these two datasets; the others are all at least one order of magnitude higher. Thus, we can conclude that the amount of the training dataset that should be used is dependent on the CV: If the CV is relatively low, but above 0.01, then around 5% of the training dataset could be used to generate weights and biases. (The same proportion would apply if the training and testing CVs are very different from one another, as was in the previously mentioned case of *S3DP*.) If, however, the CV is an order of magnitude lower than 0.01 – as is the case with *yf17Temp* – then a higher proportion of the training dataset should be used to generate weights and biases. As the CV continues to decrease, a larger portion of the training dataset should be used to generate weights and biases.

VI. Related Work

Several works have looked into reducing the training times involved in machine learning. For example, Nguyen and Widrow [8] describe a method involving initializing adaptive weights in a two-layer neural network and show that, on one example, it significantly reduces the training time of a neural network from two days to just four hours. Thomson et al., meanwhile, [9] propose using unsupervised clustering in a program's feature space to significantly reduce the amount of time required for training in a machine learning-based compiler. Laurent et al. [10] looked into using batch normalization in recurrent neural networks and found that one such effect of doing so was reduced training times.

Work has also previously been done to attempt to reduce the training time of the CAE. Liu et al. [7] looked into the possibility of using transfer learning with the CAE and found that doing so could reduce training times without significantly increasing the error in most cases.

VII. Conclusion

In this paper, we evaluate the performance when different proportions of a training dataset are used for generating weights and biases to compress a testing dataset. We conduct experiments on 18 datasets and categorize them based on their results. From this categorization, we can find that if a dataset performed well on the compression autoencoder when the full training dataset was used to generate weights and biases, then it will likely also perform well when only a portion of the training dataset is used. The size of the training dataset (relative to the size of the testing dataset) that should be used depends on the value of the CV. We also find that, in cases where the training and testing portions of the dataset have very different CVs, reducing the size of the training dataset will likely improve the prediction accuracy. By showing how the training time can be significantly lowered when reducing the amount of the training dataset used, and displaying how this can still yield results with suitable accuracy, we alleviate one of the major limitations of the compression 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] Di, Sheng, and Franck Cappello. "Fast error-bounded lossy HPC data compression with SZ." 2016 ieee international parallel and distributed processing symposium (ipdps). IEEE, 2016.
[2] Leibovici, Thomas. "Taking back control of HPC file systems with Robinhood Policy Engine." arXiv preprint arXiv:1505.01448 (2015).
[3] Zhang, Jialing, et al. "Efficient Encoding and Reconstruction of HPC Datasets for Checkpoint/Restart." 2019 35th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2019.
[4] Lu, Tao, et al. "Understanding and modeling lossy compression schemes on HPC scientific data." 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2018.
[5] Liu, Tong, et al. "High-Ratio Lossy Compression: Exploring the Autoencoder to Compress Scientific Data." IEEE Transactions on Big Data (2021).
[6] Baldi, Pierre. "Autoencoders, unsupervised learning, and deep architectures." Proceedings of ICML workshop on unsupervised and transfer learning. 2012.

[7] Liu, Tong, et al. “Exploring transfer learning to reduce training overhead of HPC data in machine learning.” 2019 IEEE International Conference on Networking, Architecture and Storage (NAS). IEEE, 2019.

[8] Nguyen, Derrick, and Bernard Widrow. “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights.” 1990 IJCNN International Joint Conference on Neural Networks. IEEE, 1990.

[9] Thomson, John, et al. “Reducing training time in a one-shot machine learning-based compiler.” International workshop on languages and compilers for parallel computing. Springer, Berlin, Heidelberg, 2009.

[10] Laurent, Csar, et al. “Batch normalized recurrent neural networks.” 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2016.

[11] “NEK5000 Documentation.” 27 July 2015, nek5000.mcs.anl.gov/files/2015/09/NEK_doc.pdf.

[12] “Flash Center for Computational Science.” FLASH CENTER, flash.uchicago.edu/site/.

[13] “MCERI: Model Calibration and Efficient Reservoir Imaging.” Texas A&M University, mceri.engr.tamu.edu/.

[14] “GROMACS Benchmarks.” 2015, gromacs.org/About_Gromacs/Benchmarks.

[15] “Scientific Data Reduction Benchmarks.” Exascale Computing Project, sdrbench.github.io/.

[16] 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. 366379.

[17] P. Lindstrom, Fixed-rate compressed floating-point arrays, IEEE transactions on visualization and computer graphics, vol. 20, no. 12, pp. 26742683, 2014.

[18] 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. 730739.