Exploring and Implementing DeepSqueeze

Michail Xydas
mikexydas@gmail.com
University of Athens
Athens, Greece

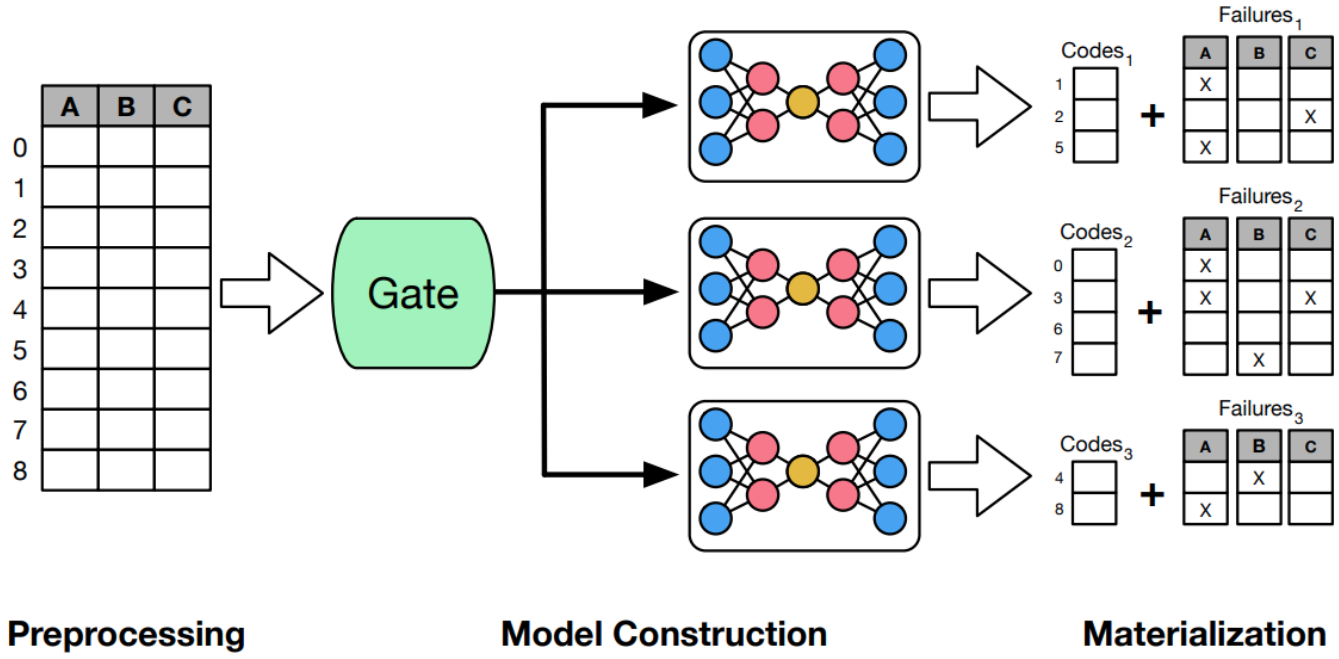

Figure 1. The full Deep Squeeze compression pipeline from preprocessing (*Sec. 1.2*), architecture (*Sec. 1.1*), and materialization (*Sec. 1.3*). *Figure is from the original paper [10].*

Abstract

DeepSqueeze [10] is a deep learning technique used for table compression. It utilizes correlations between columns that traditional compression algorithms, like gzip [3] and parquet [1], do not take into account. They employ the autoencoder architecture [9] to capture the correlations and project each row to a smaller dimension space.

We reverse-engineer this implementation managing to achieve similar results confirming its reproducibility on the 2 datasets they used. Then, we make additions and improvements like experimenting on a new real use-case dataset, improving the materialization step, and tuning the architecture hyperparameters in our attempt to reproduce and improve the compression ratio, our main evalutation metric.

Keywords: table compression, dimensionality reduction, autoencoders, reproducibility

1 Introduction

In recent years, we observe a log everything mentality from environmental sensors in the wild to system logs in the cloud. This leads to huge file sizes that lead to high storage costs, especially if we consider much-needed backups. The authors observed, that while there are many efficient compression algorithms, none of them take into account the high correlations between the columns in these specific tables. The well-known gzip algorithm is a general-purpose tool that does not consider the file type. Following, parquet can serialize and compress table structures, but it compresses each column separately assuming that they are independent of each other. Finally, Squish [8] is a representation learning technique like DeepSqueeze, however, its implementation scales exponentially, forbidding us from using it on large tables.

1.1 Achieving Dimensionality Reduction

DeepSqueeze can be described as a compression algorithm that finds a lower-dimensional representation for each table row, guaranteeing a user-defined error threshold while remaining scalable. The authors observed that highly correlated rows should be close in a semantic space of lower

dimension. This means that there is a function f that encodes each row to that space.

$$f : R^n \rightarrow R^m, n >> m$$

$$codes = f(X) \tag{1}$$

where n is the number of columns of our original table X and m the size of the lower dimensional representations named *codes*. This function f must also be invertible to be able to retrieve the original table X through

$$X = f^{-1}(codes) \tag{2}$$

We can think of *Eq.1* as the compression part and *Eq.2* as the decompression.

However, finding the invertible function f analytically is an intractable problem, so we attempt to approximate it using an autoencoder architecture.

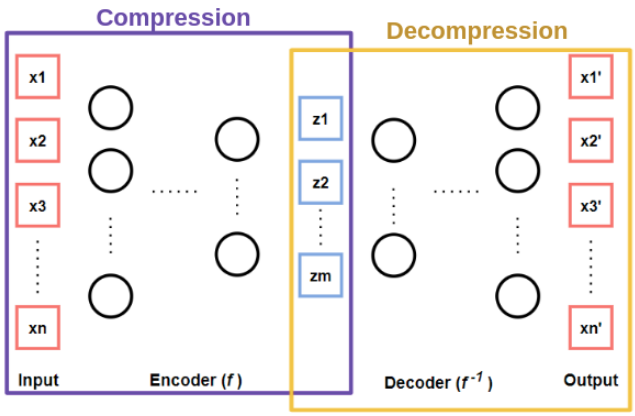

Figure 2. The auto-encoder architecture in the compression-decompression setting. We input one row of n columns, compress it to a size of m and then decompress it to the original size n.

In *Figure 2*, we input a row that consists of n columns, compress it to get a *code* of size m, and then reconstruct it back to its original size. The autoencoder is trained end-to-end by calculating the gradients through back-propagation and performing a gradient descent step iteratively. Our goal is to minimize the reconstruction loss between the input X and the reconstruction X'. The authors have chosen the Mean Squared Error (MSE) loss but we experiment with other losses too.

One could compare the above method of finding the lower-dimensional representations with the PCA method. Actually, DeepSqueeze would approximate the PCA representations if we removed all the activation functions, making our model linear. However, by employing the activation functions we create a model that captures non-linear correlations too.

1.2 Preprocessing

Before, training the autoencoder we must first preprocess our input table. There are two preprocessing steps. First, we scale each numerical column independently on the $[0, 1]$ range. Second, we perform quantization.

Quantization maps all the continuous values of our original table X to discrete ones that have a constant interval between them. The interval is defined by the error threshold given by the user and can be different for each column depending on the need for precision. For example, the values [0, 0.4, 0.26, 0.9, 1] with an error threshold of 10% will be mapped to [0.1, 0.3, 0.3, 0.9, 0.9]. The quantization steps aim at making the task of the autoencoder much easier since the number of discrete outputs is greatly reduced. On the experimentation part we have a common error threshold for all columns. The authors of the original paper have made the same assumption too.

We have to note that the scaling decision seems deceptively unimportant. However, if a dataset contains outliers it will greatly impact the precision of the decompressed file. For example imagine a column with all of its values being around 1, and a huge outlier of value 100. With an error threshold of 1% all the values except the outlier will be quantized at the first bucket [0, 2], the outlier will be mapped in bucket [98, 100] and all the other buckets will remain empty. Intuitively, in this example DeepSqueeze considers all the values except the outlier equivalent which probably is not the case.

1.3 Materialization

One main problem with deep learning and in our case the autoencoder architecture is the lack of guarantee of how well it performs. We must be able to guarantee an error upper bound that the users define depending on their use case.

DeepSqueeze offers that guarantee by the final step of the pipeline, materialization. We first pass all the *codes* through our trained decoder. We then compare the reconstructions with the quantized table that we trained our autoencoder on. Each reconstruction that does not match the exact input value is considered a "failure" and is stored in a separate data structure. In fact, the data structure that stores the failures is the one that takes up most of the size of the final compressed file. This makes it a nice target for improvements and experimentation in my custom implementation.

1.4 Mixture of Experts

Instead of having just one autoencoder, we can have a set of them with the assumption that each one will be trained on semantically different partitions of the table. This architecture is called a Mixture of Experts [11]. The partitioning is performed by a trainable gate that is trained end-to-end like the single autoencoder architecture.

1.5 Evaluation Metric

We must now define a metric that will be used to evaluate and compare the DeepSqueeze method with other compression techniques. Traditional metrics like accuracy, f1-score, etc. do not apply in this case. Since we are implementing a compression algorithm, our evaluation metric is the final size of the compressed table, normalized by the size of the original table:

$$CompressionRatio = \frac{Decoder + Codes + Failures}{OriginalSize} \quad (3)$$

Compression ratio is our main evaluation metric with the goal of minimizing it.

2 Implementation

Having briefly presented the main ideas behind each step of the compression pipeline, we showcase some details of the implementation in our attempt to reproduce the paper. Since there was not a public implementation and we were not able to contact any of the authors, we had to make a series of assumptions concerning the autoencoder architecture, the hyper-parameters, and the materialization steps.

2.1 Simplifications

Since this is a time-constraint implementation, we had to simplify the input types. We only allow numerical columns and not categorical while the original authors supported both types. We consider that this simplification does not affect the main parts of their implementation, especially if we consider that 2 out of the 5 datasets they experiment on, consist only of numerical columns (Corel [2] and Monitor[5]).

Also, as stated above we assume that all the columns have the same error threshold, which is not necessary. This simplification is made by the authors too.

2.2 Datasets

In order to evaluate DeepSqueeze, we need to perform experiments on tables of appropriate size. As stated above these tables should consist of numerical columns only. We use 2 datasets included in the original paper:

- Corel [2], consists of image color histograms that divide the color space to 32 subspaces
- Monitor [5], an analytical benchmark for machine-generated log data, mainly used for query execution benchmarking. Due to hardware constraints, we had to get a random sample version (20%) of the dataset.

2.3 Autoencoder Architecture and Hyper-parameters

Above we presented the main idea which explained why the autoencoder architecture fits our needs. In order to successfully implement an autoencoder, we must first define some parameters. The authors suggest a depth of 2 hidden layers and a width of 2 times the number of columns for the encoder and the decoder. Concerning the loss function, they propose the MSE loss for numerical columns which attempts to bring the reconstructions as close to the original values as possible. The Adam optimizer is used with a learning rate of 1e-4. Finally, we train the network for just one epoch since it always converged in the 3 datasets presented below. We implement the above using the PyTorch deep learning framework.

Two hyper-parameters have not yet been defined. These parameters are the batch size and the size of the code. Since these hyper-parameters are dataset dependent, the authors propose the usage of Bayesian Optimization to automatically define them. Given a space spanned by the batch size and the *code* size we attempt to find the point that maximizes a reward function. This function takes as input the hyper-parameters and outputs the negative of the final compression ratio (we get the negative since B.O. maximizes the function). We use the bayesian-optimization python package [6].

2.4 Materialization

The final step of our compression pipeline is the materialization step, where we create the final compressed file. The final file should be self-sufficient, meaning that we will only need the file and the decompression utility to get back the original table. In order to achieve that, the final file must include the decoder state we get from PyTorch, the *codes* that we will input to the decoder, the failures, and the scaler that will transform the range of each column from $[0, 1]$ to the original scale.

Following the paper, the failures are created using the following equation:

$$Failures = Q(X) - Decoder(Encoder(Q(X))) \quad (4)$$

where Q is the quantization process and X the original table. Then both the codes and the failures are compressed using parquet since they have a table structure. Finally, we store all 4 components (decoder, codes, failures, scaler) in a single directory which we then compress using gzip to get the final compressed file.

3 Additions

The *Implementation* part we presented above focuses on the reproduction of the original DeepSqueeze paper. In the second part, we will focus on small additions in an attempt to complete the paper with more experiments and studies and possible improvements.

3.1 Berkeley Dataset

We propose the addition of the *Intel Berkeley Research Lab Sensor Data* [4] which consists of environmental measurements (eg. humidity, luminosity, voltage) of 54 IoT sensors from 2/28/2004 - 4/5/2004. In total it contains 2.3 million readings (rows) and 8 numerical columns. The importance

Table 1. Information of all the datasets used in our experiments.

Dataset	Rows	Columns	Size
Corel	68K	32	19MB
Monitor (20%)	4.4M	18	615MB
Berkeley	2.2M	8	105MB

of this dataset is two-fold. First, it showcases a real-world example that DeepSqueeze excels at, while we would not meet the other two datasets, Corel and Monitor, in real-world use cases. Second, it emphasizes the need for an efficient table compression algorithm. This dataset only includes measurements of 54 sensors in 36 days and takes up 105MB. We can imagine applications that have many more sensors and run for greater duration and we can easily see how storage and backup costs increase fast.

The Berkeley dataset contained one "date" column and one "time of the day" column. Instead of removing them, we decided to transform them into a numerical type. The date changed to a counter of the number of days from the first measurement of 2/28/2004. The time column changed to a counter of minutes from midnight 00:00. This allows the assumption that all columns are numerical to remain valid, without losing any information.

3.2 Failure Storing

The failures are stored in a table data structure which in turn is compressed using parquet. Finding a way to "help" parquet achieve a high compression ratio is crucial since failures take up most of the space in the final file. One straightforward way to help parquet is to decrease the total number of discrete values in the failures table.

Taking under consideration the continuous output of the autoencoder, Eq. 4 will generate a huge number of unique values in our failures table. For example, consider a target value of 0.65. The autoencoder might output 0.652 getting a failure value of -0.002. However, the difference is minuscule, and more importantly, since the target value is already quantized in preprocessing, it does not mean that the result is a value closer to the original true value. Also, our MSE loss will not penalize this failure since it focuses on larger failures due to the squaring term of MSE.

We propose two different solutions to decrease the failure table compressed size.

First, we need to find a new training loss that focuses more on fixing minuscule failures than MSE does. A trivial choice is using the Mean Absolute Error (MAE) as our loss function. MSE due to the squaring term, penalizes much more big failures ignoring small ones, while MAE attends linearly to both, small and big, target-prediction differences. However, using only MAE loss will slow down our convergence. We deal with the slow down by having a loss schedule that linearly reduces the weight of the MSE loss and increases the MAE weight, as training progresses.

One second idea is to change *Eq.4* in a way that greatly decreases the number of distinct values in the failure table. We achieve that by performing post-quantization on the decoder outputs too.

$$Failures = Q(X) - \mathbf{Q}(Decoder(Encoder(Q(X)))) \quad (5)$$

Notice the addition of the **Q**uantization of the *Decoder* outputs in *Eq.5*. Since our autoencoder is trained on the quantized table we do not gain anything by not quantizing its output too. In contrast, by quantizing we make the parquet compression ratio of the failures much lower since the number of distinct values decreases significantly.

4 Experiments

4.1 Setup

All the experiments presented below were run on my personal desktop using an Nvidia RTX 2060 GPU. We note that using a GPU does not offer a substantial speed-up when compared to CPU compression times. We welcome this result since most users would not expect a GPU as a requirement to compress their table files.

Since we use a deep learning architecture, the initialization of the weights can have a major role in the final compression ratio, especially since our autoencoders are relatively small [7]. There are two possible ways of dealing with the initialization dependence. Set a constant seed to PyTorch and get a deterministic weight initialization or repeat each compression pipeline many times and report back the mean and the standard deviation. We are not certain what the original authors have chosen, but we selected the second approach of repeating each compression 10 times since it is within our time budget.

4.2 Original Implementation

We first present the results of our reverse-engineered implementation of the DeepSqueeze method without any of the proposed additions or improvements. We use the Corel dataset as a reference since it is the only one that is completely unchanged. Monitor has been randomly sampled due to hardware constraints and Berkeley is a new dataset proposed by us.

In *Figure 3*, we can see that our implementation is marginally lacking when compared to the results that the authors of DeepSqueeze. We repeated 10 times the compression pipeline and reported the mean in *Fig. 3*, however, we noticed a high standard deviation too. This means that our current pipeline is too prone to weight initialization noise, making it unstable. This makes us assume that the authors have not explicitly stated some implementation details, that are important to achieve their results.

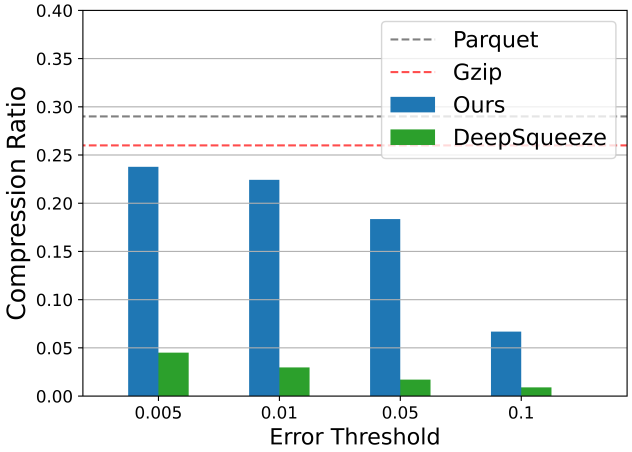

Figure 3. Corel compression ratio results of the original implementation (blue) without adding any improvement on failure storage.

Also, an unexpected result was that our baseline parquet compression ratio is slightly higher than the one reported in the original paper, shown in *Table 2*.

Table 2. Compression ratios of the baselines, gzip and parquet.

Algorithm	Original	Ours
Gzip	0.26	0.26
Parquet	**0.24**	**0.29**

In *Table 2*, we can see that while the gzip baseline has the same compression ratio on both implementations (original and ours), the parquet one has a great difference with ours being higher. We experimented with several parquet engines and compression algorithms but were unable to achieve the original paper results. Since the parquet compression has a major role in compressing the failures data structure, we expect that this will slightly impact our DeepSqueeze results too. The parquet parameters we selected were the *pyarrow* engine and the *brotli* compression algorithm.

Having established a preliminary implementation, we continue by finding details that can be optimized, especially in improving the storage of the failures data structure which takes up most of the final size

4.3 Improving Failure Storage

As presented in *Section 3.2*, we experiment with two failure storing improvement techniques. First, we propose a loss schedule focuses more on fixing small magnitude errors towards the end of the training. Second, we propose the addition of a post-quantization step when we calculate the failures.

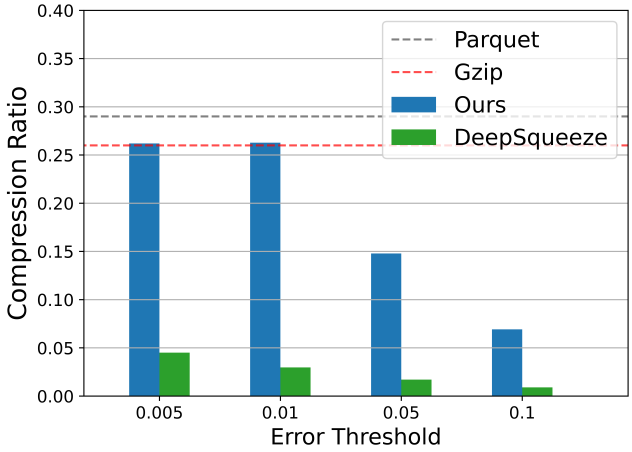

Figure 4. Corel compression ratio results using a loss schedule that linearly transitions from MSE to MAE loss.

While the loss schedule results (*Figure 4*) are more stable with less variance, we are still failing to match the results of the DeepSqueeze paper by a significant margin.

One possible cause of failure is our attempt to make a neural network to output discrete exact-match values. This goes against the capabilities of the neural network so we must focus on improving a different part of failure storing. Specifically, we propose *Equation 5* which greatly reduces the number of unique failure values by applying quantization on the decoder outputs.

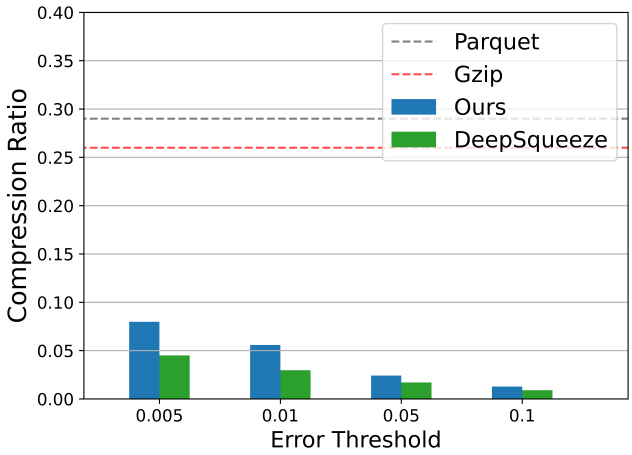

Figure 5. Corel compression ratio results with post quantization applied on the decoder outputs.

Using this quantization trick we achieve our best result so far which is close to the results of the original authors. We cannot be certain of the comparability of our results since the authors might be keeping the best (minimum) compression ratio of each error threshold instead of the mean or have a

fixed seed value that happens to work well. In addition, we were not able to infer the parquet implementation that they used.

We also note that while the authors do not explicitly state that they are performing post quantization, we assume that they have included post quantization or a similar trick in their pipeline. Unfortunately, we were not able to contact any of the authors via email to get a certain answer.

In *Figure 7*, we can see compression ratios of all the techniques (original without any additions, loss schedule, post quantization, and paper results) on all of our datasets (Corel, Monitor, Berkeley). Notice that we do not have paper results on the Berkeley dataset since it was a dataset proposed by us.

One interesting observation is the better performance of our pipeline with post-quantization on the Monitor dataset when compared to the paper implementation. This might be caused by the random sampling we have to perform on the dataset due to memory issues. However, we repeated the experiment with many random partitions of the dataset, observing similar results. This might suggest that the authors have not used post quantization to reduce the failures storage but a different idea that we were unable to infer from the original paper.

4.4 Examining Scalability

While compression ratio is the main evaluation metric, we should also take into account the time requirements of our implementation. Since we are proposing a compression algorithm, if the execution time is much more time-consuming than the traditional gzip and parquet choices, then our implementation can only be examined from a theoretical viewpoint and not in an actual utility that one could use when making storage decisions.

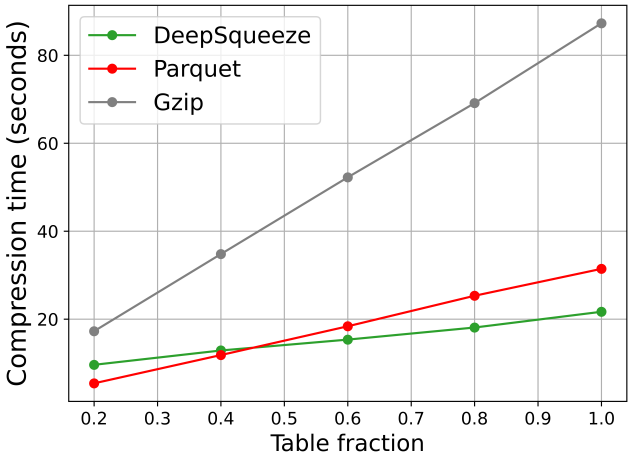

Figure 6. Compression time of DeepSqueeze, gzip and parquet on increasing sample sizes of the Monitor table.

In *Figure 6*, we can see that DeepSqueeze runs faster than gzip and its time increases logarithmically as the size of the table increases. Intuitively, this might not make sense since DeepSqueeze has to perform a pipeline of reading, preprocessing, training, and failure calculating. However, we achieve a great speed-up by training the autoencoder on a sample of the full table, which will not cost us in compression ratio if the table is highly correlated. As a result, as the table increases in size, only the materialization step takes more time, which is performed by highly optimizable and parallelizable matrix operations.

5 Conclusions

We presented an attempt at reproducing and improving the Deep Squeeze paper. Throughout the implementation, we had to make some decisions and assumptions that could lead to possible performance variations between the original results and the reproduction results. Overall, DeepSqueeze is an interesting approach at dealing with compression of huge tables, a problem that appears often due to our log everything mentality. It makes good use of Neural Networks exploiting the universal approximation theorem at its fullest.

Concerning our implementation, one question which remains unanswered is the difference in parquet compression ratio between this report and the original paper. This difference has a great impact not only on the baseline but on the DeepSqueeze implementation itself since we use parquet to compress the failure data structure.

There are still possible extensions, like studying the compression ratios of an online compression version. Also, the outlier problem presented in *Section 1.2* should be studied since its solution is not trivial. Finally, an interesting idea would be to use the same architecture but instead of compression, perform missing value restoration. We believe that in the case of filling missing values the Mixture of Experts architecture could offer significant improvements since the task is usually harder.

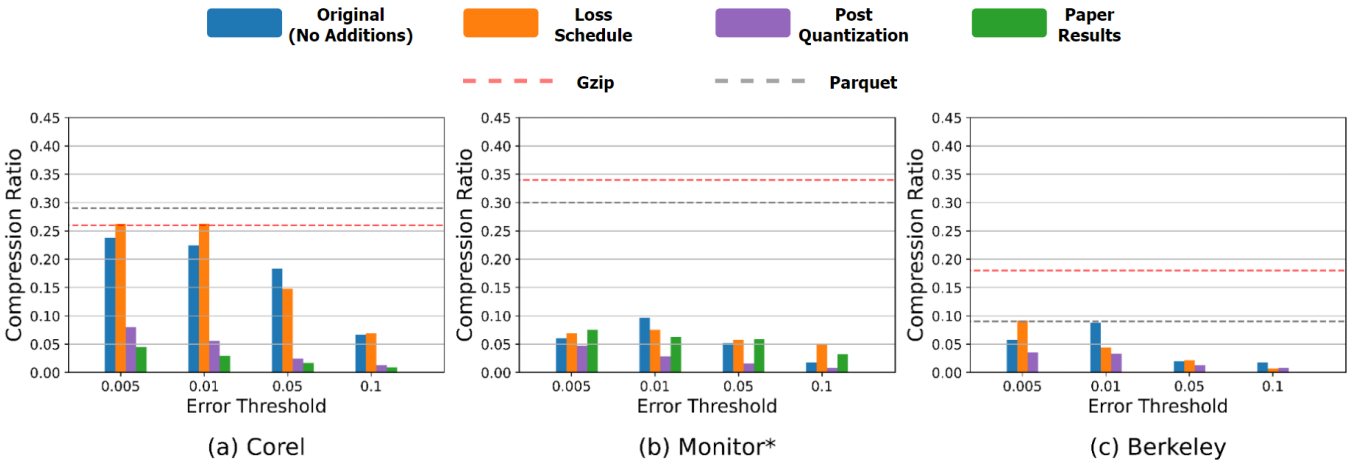

Figure 7. The results of all the compression methods presented above on all of our datasets. *We have randomly sampled Monitor dataset to 20% of its original size.

References

[1] [n.d.]. Apache Parquet. https://parquet.apache.org/

[2] [n.d.]. Corel Image Features. https://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html

[3] [n.d.]. Gzip - GNU Project - Free Software Foundation. https://www.gnu.org/software/gzip/

[4] [n.d.]. Intel Lab Data. http://db.csail.mit.edu/labdata/labdata.html

[5] Andrew Crotty. [n.d.]. crottyan/mgbench. https://github.com/crottyan/mgbench

[6] fernando. [n.d.]. fmfn/BayesianOptimization. https://github.com/fmfn/BayesianOptimization

[7] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. *arXiv:1803.03635 [cs]* (March 2019). http://arxiv.org/abs/1803.03635 arXiv: 1803.03635.

[8] Yihan Gao and Aditya Parameswaran. 2016. Squish: Near-Optimal Compression for Archival of Relational Datasets. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1575–1584. https://doi.org/10.1145/2939672.2939867

[9] G. E. Hinton. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (July 2006), 504–507. https://doi.org/10.1126/science.1127647

[10] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1733–1746. https://doi.org/10.1145/3318464.3389734

[11] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *arXiv:1701.06538 [cs, stat]* (Jan. 2017). http://arxiv.org/abs/1701.06538 arXiv: 1701.06538.