DeepSqueeze: Deep Semantic Compression for Tabular Data

Paper #487

ABSTRACT
Efficient data compression has become more important than ever with the rapid proliferation of large datasets. Traditional columnar compression techniques (e.g., dictionary encoding, run-length encoding) have proven highly effective in modern OLAP systems, but they typically compress individual columns without considering potential relationships among columns, such as functional dependencies and attribute correlations. Semantic compression techniques, on the other hand, are designed to leverage such relationships to store only a subset of the attributes necessary to infer the others, but existing approaches cannot effectively model complex relationships among more than a few attributes at a time.

We propose DeepSqueeze, a novel semantic compression framework that uses deep learning models to efficiently capture sophisticated relationships within tabular data. DeepSqueeze captures attribute relationships by directly mapping tuples to a lower-dimensional representation through the use of deep autoencoders. DeepSqueeze supports guaranteed error bounds on individual attributes and can work productively in conjunction with state-of-the-art columnar compression formats. Our experimental studies using real-world datasets demonstrate that DeepSqueeze can achieve up to almost 4x reduction in data sizes compared to what state-of-the-art alternatives can achieve.

ACM Reference Format:

1 INTRODUCTION
Across almost all domains, datasets are continuously growing both in terms of size and complexity. This is primarily due to the fact that practitioners realize the importance and effectiveness of making high-level decisions that are supported by past data. Therefore, in a wide variety of applications (e.g., autonomous vehicles, IoT deployments, web applications), it is now commonplace to “log everything” with the hopes that the data will be valuable at some future point.

However, as dataset sizes continue to increase, the cost of storage, transmission, and query processing in order to run analytics becomes a core challenge. Therefore, unsurprisingly, a wide range of compression techniques are commonly used, including byte-level (e.g., gzip, Snappy, and Brotli) as well as columnar compression (e.g., dictionary compression, run-length encoding, and delta encoding). Although effective in some scenarios, these techniques are unable to leverage the relationships that exist among attributes in many highly-dimensional datasets.

In many cases, correlations between attributes exist, since a single row often represents an entity or event. For example, in an IoT deployment, there might be correlations between the reading time and the temperature of a motion event since temperatures are often higher during the daytime.

Semantic compression takes advantage of such relationships between attributes in an attempt to reduce the storage footprint of the dataset. More specifically, semantic compression avoids having to store attribute values that can be accurately predicted from others. For example, a semantic compressor would avoid storing the state associated with a street address and instead store only the zip code since a mapping exists between zip code to state.

Although a number of semantic compression algorithms have been previously proposed [11, 18, 24, 25], these are unable to capture complex relationships between the attributes. For example, Squish [18] and Spartan [11] rely on heuristics-based methods to build candidate compression models using only a small sample of the input data. Consequently, they may miss complex relationships that are not captured by the candidate models.

Therefore, we present DeepSqueeze, a new semantic compression framework for tabular data that is able to effectively leverage relationships between attributes in order to compactly compress data. At its core, DeepSqueeze uses a series of deep learning models (in particular deep autoencoders) that make it possible to learn and leverage the complex relationships between attributes in order to compress data. As we show in our experiments, our deep learning models, combined with efficient encoding schemes and materialization techniques (for incorrectly predicted attribute values) allow us to outperform existing state-of-the-art semantic and columnar compression techniques by up to almost 4x on several real-world datasets.

Moreover, many of the previously mentioned applications can tolerate some form of imprecision. For example, cell phone geolocation data does not necessarily need to be stored...
in a lossless manner since (1) typical end applications usually only require approximate values (e.g., find the neighborhood with the highest call volume); and (2) other forms of imprecision are often present in the data (e.g., sensor hardware limitations). Thus, we also explore how to natively incorporate lossy compression into our compression algorithm.

In summary, we make the following contributions:

- We propose DeepSqueeze, a new semantic compression framework capable of capturing and leveraging highly non-linear relationships by using deep learning models in order to efficiently compress data.
- We introduce several optimizations that make DeepSqueeze "columnar compression-aware", allowing us to further compress the output of our models by applying well-known columnar compression techniques.
- We show that DeepSqueeze is able to automatically pick appropriate hyperparameters as well as divide the input data into partitions that are trained by different models to further improve performance.
- Our experimental results show that DeepSqueeze is able to outperform state-of-the-art compression techniques by up to almost 4x.

The remainder of the paper is organized as follows: Section 2 describes background and related work. Section 3 provides an overview of DeepSqueeze including a formal problem definition and compression/decompression. Section 4 explains data preparation steps. Section 5 outlines the types of deep learning models and methods used. The materialization of the resulting compressed files are explained in Section 6. Finally, we evaluate DeepSqueeze in Section 7 and conclude in Section 8.

2 BACKGROUND & RELATED WORK

Compression is a well-studied problem, both in the context of data management as well as in other research and applied areas. Such interest has spawned several classes of compression techniques, including byte-level compression, columnar compression, semantic compression, and deep compression. We note that many of these techniques are not mutually exclusive and can often be combined in multiple ways. For example, semantic compressors such as IttCompress [25] and Spartan [11] apply other types of compression methods such as lossless compression to achieve even better results. In the following, we describe the compression landscape and outline the key ideas behind each category of compressors.

2.1 Syntactic Compression

The most common class of compression algorithms is syntactic compression, which includes byte-level compression algorithms such as gzip [18]. These algorithms oblivious to the high-level semantics of the data and therefore do not take advantage of correlations between the columns. This family of compressors is usually lossless.

In particular, gzip uses the Deflate compression method [14], which outputs a series of blocks, corresponding to successive blocks of input data. Each block is compressed using a combination of the LZ77 algorithm [39] and Huffman coding and consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part, and a compressed data part. In the same family of compressors, Snappy [19] is another lossless compression/decompression library developed by Google. Instead of trying to achieve the highest compression ratio, Snappy aims to also optimize for compression/decompression speeds, and a high level of robustness to handle corrupted input. Brotli [8] is another open-source syntactic compression library that also uses both LZ77 and Huffman encoding that is proposed to replace gzip by offering not only a smaller compressed file size, but also faster compression/decompression times for Internet applications. Finally, LZFSE [9] uses Finite State Entropy coding and targets similar compression goals at higher compression and decompression speeds (as compared with Deflate) by using Zlib [16]. Other syntactic compressors include Zstandard [13], LZHAM [21], and LZ4 [12].

Finally, several techniques have been proposed specifically for machine learning and linear algebra operations [17, 27]. Unlike DeepSqueeze, these techniques try to balance ratio as well as runtime, leading to larger compressed representations.

2.2 Columnar Compression

With the advent of column-store databases like C-Store [36] and Vertica [26], various compression techniques were proposed including well-known techniques such as dictionary encoding, run-length encoding, and delta encoding [6, 7].

Parquet [29] is a columnar data storage format developed initially as part of the the Apache Hadoop project. Parquet uses different encoding schemes depending on the dataset and data distribution (e.g., using delta encoding for sorted numeric attributes, dictionary compression for low-cardinality string columns). Parquet is generally used as a storage format for data processing frameworks (e.g., SparkSQL [10]). As previously mentioned, though, it can also be used on top of other compression techniques (e.g., semantic compression). As we describe later, DeepSqueeze also uses Parquet to further compress the internal representation of the input dataset produced by its deep autoencoders.

2.3 Semantic Compression

Semantic compression, unlike syntactic or columnar compression, aims to leverage high-level knowledge about the data to improve its compression effectiveness. While there
are several semantic compression methods, the most common methods leverage correlations between the columns of a tabular data to omit redundant values.

Fascicles [24] is one of the first works in semantic compression. It searches for rows with the maximum number of mergeable attributes and clusters them into “fasicles”. Similarly, ItCompress [25] is an iterative compression framework that aims to select a set of “representative rows”. Each tuple in the data is then assigned to one representative. After each assignment, the representative rows are updated so that the number of erroneous predictions is minimized. This iteration continues until the algorithm reaches a convergence point where no more improvement is possible.

The more general idea of Minimum Description Length (MDL) [37] is closely connected with Fascicles and ItCompress. Although the focus of MDL is discovering the most important patterns in a dataset, it is shown that such patterns are equivalent to patterns that are able to represent the data in the most succinct manner. Based on this, Fascicles and ItCompress can be thought of as special lossy versions of MDL-based approaches that attempt to discover the most important patterns (or representatives in the case of ItCompress) that are able to describe as many rows as possible.

On the other hand, Spartan [11] employs data mining techniques to find Classification and Regression Trees (CaRT) that are used to learn relationships between the attributes, possibly eliminating the need to store entire columns.

Entropy compression [32] is another high level compression technique that allows queryable lossless compression of relations. Although these techniques leverage properties about relational data (e.g., unordered), they are unable to model highly non-linear relationships between attributes to further compress data since they only consider pairwise correlations between attributes.

Finally, Squish [18] uses a combination of Bayesian Networks and Arithmetic Coding to capture the dependencies among attributes. As the authors argue, if all of the given attributes are categorical and the learned Bayesian Network is accurate, then Squish is optimal. However, learning an accurate Bayesian Network itself is an NP-hard problem and therefore they limit the problem into learning such a model only if all attributes are categorical.

### 2.4 Deep Compression

Since deep learning has the unique ability to learn complex relationships in data, it is a natural candidate to use in order to improve compression. So far, deep learning with convolutional neural networks (CNNs) has been successfully applied to image compression [28, 31, 33], as well as to other related problems (e.g., denoising). However, to the best of our knowledge, DeepSqueeze is the first compression method that applies deep models to the task of compressing tabular data. Importantly, our proposed techniques can handle datasets with both categorical and numeric values with strict per-attribute error thresholds, unlike image data that is purely numeric. Additionally, we propose several optimizations that are specific to relational data, such as different failure encoding and materialization optimizations.

### 3 DEEPSQUEEZE OVERVIEW

Unlike prior work, DeepSqueeze uses deep autoencoders in order to learn the complex patterns that exist in real-world data for efficient compression. In the following, we formally define the problem and provide a high-level description of how compression and decompression work in DeepSqueeze. Section 4, Section 5, and Section 6 discuss the preprocessing, model, and materialization aspects, respectively.

#### 3.1 Formal Problem Definition

The input to DeepSqueeze is a relational table consisting of \( n \) rows and \( m \) attributes, each of which are either numeric or categorical along with a vector \( \hat{y} \in \{0,1\}^m \) of size \( m \) that specifies the type of each attribute.

In addition to the input data, the user can specify an error vector \( e \) of size \( m \) with values ranging between \([0,1]\) that represents the maximum acceptable error for numeric attributes. The absolute tolerable error is computed by pointwise multiplication of \( e \) and a vector \( w \) that represents the width of each attribute. Let \( n_n \) and \( n_c \) denote the number of numerical and categorical attributes, respectively. \( w \) is defined as \( w_i = max(N_i) - min(N_i) \) for \( i \) in \([0,m_c]\) where \( N_i \) is the \( i \)’th numerical attribute and \( w_i \) is the difference between the maximum and minimum values taken by that attribute. For example, if an \( \text{Age} \) attribute has values in the range \([2, 102]\) and the corresponding error tolerance value is 0.01, the absolute difference between an actual and predicted age must be at most \( 0.01 \times (102 - 2) = 1 \) year.

In this work, we consider lossy compression only for numerical attributes. Thus, we rebuild the categorical attributes without any error. However, it is also possible to support lossy compression for categorical attributes by allowing a bounded number of values to be incorrectly classified.

#### 3.2 Compression

In the following, we provide a high-level overview of DeepSqueeze’s compression process (illustrated in Figure 1) and expand upon the details in later sections.

The first step (Section 4) is to preprocess the input data into a format upon which the model can operate. Depending on the data type, we apply well-known transformations (e.g.,
dictionary encoding), as well as a special quantization technique that respects the specified error threshold for numeric attributes.

In the next step (Section 5), we build a type of artificial neural network called an autoencoder that maps the features of the preprocessed input data to a lower-dimensional representation. Autoencoders are similar to other dimensionality reduction techniques, but they have the ability to learn nonlinear relationships, which are common in real-world datasets. The unsupervised training process proceeds iteratively over the dataset until convergence, and we developed several optimization techniques to tune model construction. Again, unlike traditional machine learning settings, our goal is to overfit the model to the input data in order to produce the smallest compressed representation possible.

One way of overfitting the training data is to create a complex model capable of capturing all relationships in the dataset, whereas an alternative approach involves building multiple simpler models, called a mixture of experts, that are specialized for certain similar partitions of the data. Figure 1 shows three specialized models, and rows are routed to models by a gate that learns the best partitioning of the data during training.

Finally, the materialization step (Section 6) uses the trained autoencoder to generate the lower-dimensional representation, labeled “Codes” in the figure. Each of these codes represents a single row, and is much smaller than the row. We also have the failures in order to respect the error threshold for lossiness. Additionally, we must also store the decoder half of the model used to decompress the codes.

3.3 Decompression

To decompress the data, we essentially perform the inverse of the compression process. More specifically, we feed the codes through the saved decoder to reconstruct a lossy version of the original data. Then, we compare each reconstructed row to the materialized failures and replace any errors with the correct value. After a final step of inverting the preprocessing, the resulting output is a decompressed version of the original dataset that respects the specified error thresholds.

4 PREPROCESSING

As mentioned, the first step of the compression pipeline is to preprocess the data, which ensures that the data is in a state that can be used to train our models. This section describes the preprocessing techniques that DeepSqueeze applies for each type of attribute.

4.1 Categorical Attributes

For categorical attributes with reasonably small cardinalities, it is often beneficial to apply dictionary encoding to minimize the space required to store the values. Moreover, in order to make the attributes suitable for model training, the values need to be replaced with numbers. This is because (1) the model training requires them to be integers and (2) they need to be in the range \([0, \# \text{ of possible outcomes}]\).

Therefore, in the preprocessing phase, we apply dictionary encoding to replace values of such attributes with indices. For example, an attribute that can take values \(A, B, C\) will be replaced with \([0, 1, 2]\), respectively.

For attributes with many distinct values (e.g., strings with many values, primary keys), DeepSqueeze automatically falls back to existing syntactic and/or columnar compression techniques. However, in particular cases when the distribution of the values is skewed, the low frequency values can be ignored while training such that the model is only trained on the top \(k\) most frequently occurring categorical values and fails at predicting extremely rare values. Since reducing the number of possible output values for categorical attributes can dramatically reduce the number of parameters in the...
model, these failures are offset by the reduction in model size.

4.2 Numerical Attributes

Since numerical attributes can tolerate some user-specified loss, we have two options for compressing them. The first option is to allow the model to make inaccurate predictions on numerical attributes and at the time of evaluation, check to see if the amount of error is within bounds: in this case the only type of preprocessing that might be required is normalization methods such as min-max scaling. Additionally, the loss functions should be modified (e.g., truncation) to account for the error bounds.

An alternative way of handling numerical attributes is to first quantize the data by converting all numbers into integers within a range and require the model to make correct predictions only: in this method the errors are precomputed in the data that is fed to the model, therefore, the model should make error-free predictions to be considered correct.

This second approach has two benefits: (1) it makes the model much simpler, resulting in a smaller model size; (2) it makes it much easier for the model to learn, resulting in fewer mispredictions. Moreover, the output of the quantized predictions are much easier for a columnar compression algorithm to compress. All of these, ultimately, result in a smaller overall compression output.

5 MODEL

One key advantage of DeepSqueeze is the ability to capture complex relationships between attributes through its use of deep learning models. In this section, we describe the architecture of the compression and decompression models that DeepSqueeze uses to learn such relationships. Then, we describe a partitioning method that allows the network to train multiple models over distinct partitions of data to improve performance. Finally, we show how we can apply Bayesian Optimization techniques to find a set of hyperparameters that achieve a minimal compression ratio given a budget.

5.1 Basic Architecture

As mentioned, DeepSqueeze uses a type of artificial neural network called autoencoder [38] to model complex relationships between the attributes. An autoencoder, in its most basic form, is a neural network that takes as input a data point (i.e., a row of a data table), creates an internal representation, and then attempts to reconstruct the same data point. The simplest form of autoencoder is a network with an input, a single hidden layer and an output with no nonlinear activation functions between the layers. In terms of modeling capacity, such a basic autoencoder is most similar to Principal Component Analysis (PCA) dimensionality reduction technique. By adding other hidden layers and non-linear activation function between the layers, an autoencoder network is capable of learning complex (i.e., non-linear) relationships between the attributes.

An autoencoder learns two different models. The first, called an encoder, maps the data points into an r dimensional space S_r (usually with a lower dimensionality). The second model that maps the data point back to the original space is called a decoder. The number of dimensions (r) in S_r can be either enforced structurally or through regularization that penalizes the model for using many non-zero nodes in the representation layer S_r. While the task of reconstructing an input is relatively simple, an autoencoder is usually capable of creating an internal representation that captures the most important information about the data.

To achieve compression, we use autoencoders with smaller representation layers than the original input data. In other words, if the data has m attributes, the autoencoder(s) used for modeling the data must have r < m dimensions.

More formally, an autoencoder applies two types of transitions E (encoder) and D (decoder) on the input which are specified in the following equation.

\[
E : \mathbb{R}^m \rightarrow \mathbb{R}^r \\
D : \mathbb{R}^r \rightarrow \mathbb{R}^m \\
s.t. \quad E, D = \arg\min_{E, D} \|X - (E \cdot D)X\|^2
\]

As shown, the transformations respect the dimensionality requirements of the original and the representation space and minimize the squared reconstruction error. Note that in this formal definition [38], it is assumed that all of the dimensions correspond to continuous (i.e., numerical) attributes. Since, in our application, we have discrete attributes on top of the numerical attributes, the last line of Equation 1 can be modified to incorporate two types of errors: one as a result of numerical predictions, and another one as a result of categorical predictions.

The modified objective function is shown in Equation 2 where \(X_a\) and \(X_c\) are the set of numerical and categorical attributes (one-hot encoded), respectively. For categorical attributes, applying the Softmax function on top of the original transformations and then using negative log likelihood as the reconstruction loss ensures that the encoder and decoder models take categorical reconstruction errors into account. \(\alpha\) and \(\beta\) are constants that regulate the relative importance of numerical and categorical attributes in terms of space requirement.
Figure 2: Autoencoder for compressing a table with three numeric and one categorical attributes.

\[
s.t. \quad E, D = \arg\min_{E, D} \{ \alpha \cdot \|X_n - (E \cdot D)X_n\|^2 - \beta \cdot X_c \log(\text{Softmax}(E \cdot D)) \} \tag{2} \]

If there is a difference between the space requirements for two attributes of the same type, a different set of \(\alpha\) and \(\beta\) values can be applied on each attribute (i.e., \(\alpha\) and \(\beta\) would be vectors instead). Since we apply quantization (described further in Section 4) as a preprocessing step, the space requirements for the two types of attributes become similar, and therefore, the values of \(\alpha\) and \(\beta\) becomes less important (both can be set to 1).

Figure 2 represents a typical autoencoder that DeepSqueeze uses to compress data. Unlike a traditional autoencoder, the autoencoders that DeepSqueeze uses must support both numerical and categorical attributes. In this example, DeepSqueeze is compressing a dataset that has 3 numerical (marked as \(N\)) and 1 categorical (marked as \(C\)) attributes. A numerical input node maps to exactly one numerical output node. However, the categorical attributes map to more than one output depending on the number of possible values for the given attribute. For example, in Figure 2, the categorical attribute has 4 different possible values.

This modified architecture allows us to evaluate the model in a different manner for numerical and categorical attributes. Since there is no closeness relationship between the values taken by the categorical values (i.e., \textit{male} and \textit{female}), it should produce the right answer in the last layer regardless of how close or far the generated index is to the ground truth.

One of the problems that arise from using such an architecture is the large number of parameters introduced as a result of assigning one output node per each possible value for categorical attributes, leading to extremely large models.

Figure 3: Parameter sharing for categorical attributes.

To tackle this problem, we introduce a shared output layer for categorical attributes whose number of output nodes is bounded by the number of distinct values of the categorical attribute with the highest cardinality. In order to achieve the described parameter sharing, we also introduce an auxiliary layer before the shared layer. The auxiliary layer contains the same number of nodes as the number of categorical attributes plus another node that is called signal. In essence, when the decoder model attempts to reconstruct a categorical attribute, it plugs the corresponding node and a generated signal in the auxiliary layer to the shared layer. The shared layer, based on the signal and the value of the auxiliary input, decides which output node should be activated. Just like a regular categorical output layer, the shared layer applies the \text{Softmax} function on top of the nodes. The signal is simply the index of the categorical attribute that can be simply generated in the code at the training and decompression phases. In other words, the signal allows the shared layer to decide how to interpret the other value that is plugged in from the auxiliary layer.

Figure 3 depicts the last two layers of the decoder used in DeepSqueeze for the previous example (numerical attributes are not shown) where there are 3 categorical attributes to be reconstructed. The three categorical attributes have 4, 3, and 5 outputs, respectively (each denoted by \(c\) with different colors). Since there are three categorical attributes, we add three auxiliary nodes plus a single signal node to the network. The shared output layer has 5 nodes since the maximum number of distinct values taken by the largest categorical attribute is 5. The original partial network require \(4 \times 12 = 48\) parameters. However, the shared parameter approach requires \(4 \times 3 + 2 \times 5 = 22\) parameters resulting in 54\% reduction in the number of required parameters even in such a simple example.

The size and the number of layers required for the architecture is a hyperparameter that needs to be fixed before the training. An autoencoder with only two hidden layers
is theoretically capable of learning any arbitrary decision boundary [30] while a single hidden layer is able to learn any function that performs a continuous mapping from one finite space to another [23]. There are different heuristics for choosing the number of nodes in each layer. We choose $2 \times m$ number of nodes (where $m$ is the total number of attributes) in each of the hidden layers except for the representation layer but other values can also be used. The number of layers for the representation layer (code layer) is a hyperparameter that is decided at training time.

The most obvious approach for improving model accuracy (and therefore compression ratio) is to increase its size (and, in turn, its learning power). However, a larger model does not necessarily always compensate the overhead it introduces, as the increase in model size might offset the gains obtained by reduced materialized failures.

Rather than creating a larger model, another alternative is to build several smaller and less complex models for different parts of the data. The intuition behind this is that a given neural network can learn a limited class of functions that are not too complex. Therefore, if we cluster the data into smaller non-overlapping subsets, we might be able to capture the relationships using a group of simpler neural networks with lower number of layers and nodes.

One obvious approach to partitioning the data is to apply a well-known clustering algorithm like k-means, with the idea that data points grouped into the same cluster will be more easier to learn using the same model. Surprisingly, though, this intuition may not always prove true. Consider the simple example shown in Figure 4. A clustering algorithm such as k-means that works based on closeness would cluster the points that are close to each other into the same group. In the example, the oval shaped areas show a possible grouping based on closeness of points when the number of centroids is set to 2 using k-means.

An autoencoder capable of capturing the distribution of the three points in each of these areas would need to be non-linear if we force the autoencoder to use a one dimensional representation because the points do not lie on a line. However, if the points that are next to the two dashed lines were grouped together, two simple linear models could capture the one dimensional manifolds on which the data points reside. Clearly, a linear auto-encoder requires less space because it is simpler in architecture, and it is easier to train compared to a non-linear one.

5.2 Mixture of Experts

Based on this observation, instead of a standard clustering algorithm, we use a sparsely gated mixture of experts model [34] that learns how to best partition the dataset in an unsupervised fashion. At the core of this architecture is a gate (an independent neural network). The gate has as many outputs as the number of partitions (or models) specified at runtime and assigns a given row to the model that has the smallest amount of prediction error.

Figure 4 represents a logical view of the gate. In practice, however, the input rows are routed to all of the models but the gate chooses which models’ output to be used by producing a sparse vector of 1s and 0s. The original sparsely gated mixture of experts model does not require the output of the gate to be one-hot, however, in our application, because we are interested in lower compression ratio, and because each model requires extra information such as codes, we choose to use a single model per partition to save space. Each independent model is referred to as an expert meaning that the model specializes on modeling only a particular set of records. Assuming that the output of the expert $i$ is denoted by $E_i(x)$, and the gate network result is denoted by $G(x)$, then the final output of the network can be written as follows.

$$\hat{x} = \sum_{i=1}^{N_E} G(x) E_i(x)$$ (3)

In this equation, $N_E$ is the total number of experts. $x$ is the original data (e.g., the table shown in Figure 1), and $\hat{x}$ is the reconstructed output. To achieve sparsity, the gate applies the Softmax function on top of another function that is called $KeepTopK$ as described in the original paper [34]. The $KeepTopK$ function (Equation 4) itself takes two arguments: $H(x)$ which is defined in Equation 5, and $k \in [1, N_E]$.

$$KeepTopK(v,k) = \begin{cases} v_i, & \text{if } v_i \text{ is in the top } k \text{ elements of } v \\ \infty, & \text{otherwise} \end{cases}$$ (4)

The $KeepTopK$ function takes as input a vector $v$ as well as a constant $k$ and outputs another vector for which top $k$ elements are kept as-is but the other elements are replaced with $\infty$. This replacement allows the corresponding outputs of the subsequent Softmax function be extremely close to
number of distinct values taken by any categorical attribute, \(X_{c_k}\) is the ground truth set of labels for the \(k\)'th categorical attribute, and \(\hat{X}_{c_k}\) is the set of predicted labels for that attribute. In order to compute the loss, labels need to be converted into one-hot encoding first. \(Y_{c_k}\) is the one-hot encoding of the ground truth labels for categorical attribute \(k\) and \(\hat{Y}_{c_k}\) is the probability matrix generated as a result of applying Softmax function at the end of shared layer (Figure 3) for categorical attributes. \(L_c\) represents a vector with \(m_c\) elements where \(m_c\) is the number of categorical attributes. Element \(k\) of vector \(L_c\) is equal to the value computed for \(L_{c_k}\). The final loss for categorical attributes \(L\) is computed by taking the average of all \(L_{c_k}\)'s for all possible values of \(k (k \in [1, m_c])\).

\[
L_{c_k} = L(\hat{X}_{c_k}, X_{c_k}) = -\frac{1}{1 \times n} \cdot \left[ Y_{c_k} \cdot \log(\hat{Y}_{c_k}) \right] 
\]

\[
L = \frac{1}{n} \cdot \sum_{k=1}^{m_c} L_{c_k} 
\]

The values in a numerical attribute even after applying the preprocessing step preserve their closeness relationships. For example, the predicted value 5 is closer to the actual value of 4 compared to a predicted value of 7. Since our the final materialization step stores the difference between the actual and predicted value (as described in Section 6, we want our model to predict values as close as possible to the target value (ideally the same). Even if the model is not able to produce the exact actual values, we can achieve more compression if the differences are bounded by a value much smaller than the range of the attribute after preprocessing. For example, if the range of the values taken by one attribute after quantization is [0, 100] (i.e., the error tolerance is set to 0.005) requiring 7 bits for each value, then a model that produces predictions that are within ±5 of the actual value, will reduce the number of bits that are required to save the differences into 4 bits. Section 6 explains how DeepSqueeze materializes the encoded and failed rows, along with corresponding optimizations, in more detail.

Based on this intuition, the Mean Squared Error (MSE) function which is shown in Equation 7 is a good candidate for numerical attributes even after they are converted into integers. In this equation, \(n\) is the number of rows and \(m_n\) is the number of numerical attributes. \(X_n\) is the ground truth matrix for numerical attributes and \(\hat{X}_n\) is the corresponding predictions for these values. The loss is the mean value of pointwise squares of the matrix resulted by subtracting the predictions from the ground truth matrix.

\[
MSE(\hat{X}_n, X_n) = \frac{1}{m_n \times n} \cdot \frac{1}{1 \times n} \cdot \left[ X_n - \hat{X}_n \right]^{\otimes 2} 
\]
5.4 Hyperparameter Tuning

One limitation of deep models is the number of its hyperparameters. Manual tuning of these hyperparameters is often too time consuming and can result in suboptimal models. Bayesian optimization [35] is an effective method for searching the best combination of hyperparameters which works by exploring the range of possible values specified by the user in an informed search. An acquisition function predicts the next most promising candidate combination of hyperparameters that the algorithm tries on the next step. Together with the set of candidate values, the algorithm receives a budget value that specifies the number of calls to an expensive function f(x). In our case, this function is the compression part of DeepSqueeze which performs the compression on a given raw file and returns the compression ratio. The overall goal of our hyperparameter optimization procedure is to minimize the overall size of the compressed file.

The set of hyperparameters that DeepSqueeze has includes learning rate, number of nodes in representation layer, and if the mixture of expert is used, the number of experts. Algorithm 1 is a pseudocode description of the hyperparameter optimization procedure. The Compress function is the main expensive function that the optimizer needs to call to perform the compression. This function first build a network by calling BuildNetwork function. It passes hyperparameter values including representation layer size (code_size), and number of experts (num_ofExperts) to the BuildNetwork function. After the network is returned by this function, it calls the TrainAndTest by passing the original data (data), the returned network (deep_network) and learning rate (learning_rate). The called function returns a compressed version of the input data. Finally, it returns the compression ratio by calling ComputeCompressionRatio. This function divides the size of the compressed file by the size of the original file and return the result. The Compress function returns the compression ration (comp_ratio) as the final result.

The BayesianTuning function receives the Compress function, maximum number of calls (max_calls), an exploration region (region) which is the domain of values that we want to try, and the original data (data) as the arguments. This function then calls Compress as many times as the value stored in max_calls. The minimum compression ratio achieved so far is stored in the variable min. Also, all of the previous trials and sets of all attempted hyperparameter combinations are stored in history. Based on the values stored in history, MaxAcquisition function returns a set of candidate hyperparameters (stored as candidate) that maximize the chances of a better compression ratio. Finally, BayesianTuning function returns the minimum compression ratio and the best combination of hyperparameters stored in history variable.

Algorithm 1 Hyperparameter Tuning

\[
\begin{align*}
\text{Compress}(\text{data}, \text{code_size}, \text{num_ofExperts}, \text{learning_rate}) & \\
\text{deep_network} & \leftarrow \text{BuildNetwork}((\text{code_size}, \text{num_ofExperts}) \text{learning_rate}) \\
\text{compressed} & \leftarrow \text{TrainAndTest}(\text{data}, \text{deep_network}, \text{learning_rate}) \\
\text{comp_ratio} & \leftarrow \text{ComputeCompressionRatio}(\text{compressed}, \text{data}) \\
\text{return} \text{comp_ratio} \\
\text{BayesianTuning}(\text{Compress, max_calls, region, data}) & \\
\text{min} & \leftarrow +\infty \\
\text{history} & \leftarrow \text{Null} \\
i & \leftarrow 0 \\
\text{while } i < \text{max calls} & \\
\text{do} & \\
\text{candidate} & \leftarrow \text{MaxAcquisition(history, region)} \\
\text{best} & \leftarrow \text{Min(best, Compress(data, candidate.code_size, candidate.num_ofExperts, candidate.learning_rate))} \\
i & \leftarrow i + 1 \\
\text{return} \text{min, history, best} \\
\end{align*}
\]

6 MATERIALIZATION

The final step in the compression pipeline is to materialize the pieces necessary for decompression, which include the (1) decoder, (2) codes, (3) failures, and (4) expert mapping. The size of the compressed output, then, is computed as the sum of each of these individual components. In this section, we describe the different techniques DeepSqueeze applies to minimize the compressed size.

6.1 Decoder

As previously explained, an autoencoder consists of an encoder that converts a row to a compressed code and a decoder that reconstructs (an approximate version of) the original row from that compressed code. Since the encoder is needed exclusively during the compression process, DeepSqueeze discards that half of the model and stores only the decoder half by exporting the weights from each of the experts.

As a simple additional optimization, we apply a final gzip on the exported weights, which can further reduce the model size by an often small amount. Other techniques for compressing neural networks are complementary but beyond the scope of this work, and they could potentially yield a significant reduction in decoder size. However, as our experiments demonstrate (Section 7.2), the materialized decoder often represents a relatively small fraction of the overall compressed output, suggesting that optimization effort should go elsewhere (e.g., reducing the size of materialized failures).

6.2 Codes

Again, the compressed codes produced by the encoder are lower-dimensional representations of each input row. The codes are 64-bit floating point values, but 64 bits of precision
is often unnecessary for accurately reconstructing the input row. Therefore, DeepSqueeze iteratively truncates each code until the reduction in code size no longer pays for the increase in number of failures. Although we could achieve further compression by creating variable-length codes, we currently only truncate codes in increments of one byte for simplicity. For all datasets in our experiments, we were able to reduce the code sizes from 64 to 16 bits, resulting in a 4× decrease in code size.

Since floating-point values are generally difficult to compress, a final post-processing step after the truncation can further reduce compressed code size by converting them to integers. For this, we multiply the codes by the smallest power of 10 necessary to ensure that the code with the largest number of decimals is converted to a whole number, and then we cast the result to an integer type. The codes can then be compressed more effectively using standard integer compression techniques (e.g., delta encoding).

6.3 Failures
An efficient storage of failing predictions can significantly enhance the compression ratio. Therefore, we argue that the different data types require to be handled differently when they are encoded. Specifically, we not only treat categorical and numerical attributes differently, but also based on the number of outputs, separate categorical attributes into two groups of binary outputs and multi-value outputs, and treat these two groups differently.

DeepSqueeze’s compression is aware of the final columnar compression that is applied after the initial model based compression, making it different from other approaches (e.g., Squish) that apply a final round of syntactic compression (e.g., gzip). In particular, we use Parquet [29] to further compress specific compressed outputs.

6.3.1 Categorical Attributes. For categorical attributes, we simply store either a null value meaning that the prediction from the decoder is correct, or the actual value of the attribute if the prediction is incorrect. Since the model primarily makes correct predictions, then the number of nulls in the materialized failures should be relatively small, such that Parquet can efficiently compress the repeated nulls. Moreover, any systematic mispredictions can be efficiently handled with variable length encoding techniques.

A special case of categorical attributes is binary attributes. Instead of materializing nulls for correct predictions, which would increase the domain to size three (i.e., null, 0, and 1), we instead save failures as 1 and correct predictions as 0 which reduces the size of the materialized failures. Then, when decompressing, we flip the predictions whose corresponding value in the failures is 1, while keeping the predictions for 0 (whether they are 0 or 1) the same. The goal of this optimization is to produce long runs of either 0 or 1 that can then be effectively compressed using techniques like run-length encoding when the model makes many correct (or systematically incorrect) predictions.

6.3.2 Numerical Attributes. Unlike categorical attributes, numerical attributes typically have a much broader range of distinct values. Additionally, numerical attributes have an implied order, such that certain predictions can be closer to the actual values than others; categorical attributes, on the other hand, are either correct or incorrect, with no notion of closeness.

DeepSqueeze therefore leverages this “closeness” property to further compress mispredictions for numerical values. Rather than storing the correct value, as is done for categorical attributes, we store the difference between the predicted and actual values. Again, the intuition is that the predicted values will be close to the actual values, which should significantly reduce the range of values that need to be stored.

The best case is that the model’s prediction is correct, which will result in storing a value of 0 since there is no difference between the predicted and actual values. Even if the model makes mistakes, if they are systematic, say within ±1 of the actual values, the minimum number of bits required to represent the difference can be much lower than the number of bits required for saving the actual values for failures.

6.4 Expert Mapping
The last piece of information that needs to be materialized in order to decompress the data is metadata that maps rows to the correct expert decoder. However, this information is not always necessary to store in all cases.

The first and most obvious case is when the model only contains a single expert, such that all rows will be passed to this single decoder. A second case is when the exact order of rows in the original data is not necessary to maintain, such as for a relational table where order is unimportant. However, for all experiments in our evaluation, we always reconstruct the file in the correct order by materializing this expert mapping metadata.

There are two different ways we consider to store the expert mappings. One approach is depicted in Figure 1, where the codes and failures are stored with their indexes grouped by expert. In this approach, the indexes tell DeepSqueeze the order in which to reconstruct the original file, and they can often be compressed efficiently by delta encoding.

It is also possible to store all rows together in the original order with an auxiliary file to store the expert assignment label for each row, which are produced by the gate during the compression phase. During decompression, DeepSqueeze uses these assignments to choose the correct decoder to use
for each row. The expert labels can be efficiently compressed using dictionary and run-length encoding.

The choice between these two alternatives (i.e., positional indexes and expert assignment labels) must be made on a case-by-case basis, since each is better in certain situations.

7 EVALUATION

In this section, we evaluate the effectiveness of DeepSqueeze using a variety of different datasets and comparison points, including gzip, Parquet, and the state-of-the-art semantic compression framework, Squish. Our DeepSqueeze prototype is implemented in Python and uses Tensorflow’s Keras interface for building and training the deep autoencoders. We compare DeepSqueeze to Squish by using the implementation that is available from the corresponding authors.

In the following, we first describe the experimental setup and tested datasets. Then, in Section 7.2, we compare the compression ratio for DeepSqueeze to gzip, Parquet, and Squish using the tested datasets for various error thresholds. Section 7.3 compares the runtime of DeepSqueeze to other approaches. Finally, Section 7.4 presents microbenchmarks that show the impact of our proposed mixture of experts, hyperparameter tuning, and sampling techniques.

7.1 Setup

We conducted all of the experiments on a single machine with two Intel Xeon Gold 6150 CPUs (2.7GHz, 18 cores, 24.75MB L3 cache), 376GiB RAM, and two NVIDIA Tesla V100 SXM2 GPUs (16GB). In all cases, we deploy Tensorflow in RAM only mode. DeepSqueeze is implemented using Keras functional API [20] with Tensorflow 1.12.0 [22] as backbone. The number of epochs used for the training process is set to 5.

We evaluate our techniques on four real-world datasets: (1) Corel [3], which has 68k rows with 32 numerical attributes; (2) Forest Cover [2], which has 500k rows with 55 attributes, out of which 10 are numerical and the rest are categorical; (3) Census [1], which has 2.4M rows with 68 categorical attributes; (4) PerfMon [5], which has 23,468,845 records with 17 numerical attributes, and (5) Criteo [4], which has around 4.5B records with 13 numerical attributes and 23 categorical attributes. The Corel, Forest Cover, and versions of Census datasets have been used to evaluate past work. These datasets represent a wide range of inputs from all numerical to mixed and all categorical datasets, which gives a good idea of performance in different settings, but they are all relatively small. In order to test the compressor’s performance on large files, we use two datasets which have not been used in previous work: PerfMon and Criteo which are 3.3 GB and 1TB in size respectively.

7.2 Compression Ratio

The primary performance metric that we consider is compression ratio, which is defined as the size of compressed output divided by the size of original dataset. We show the results for the four real-world datasets in Figure 5, with compression
Table 1: Compression Time (Left) and Decompression Runtimes (Right) in Seconds

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Corel</th>
<th>Forest Cover</th>
<th>Census</th>
<th>PerfMon</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>1.31</td>
<td>5.78</td>
<td>2.57</td>
<td>1.08</td>
</tr>
<tr>
<td>Parquet</td>
<td>1.33</td>
<td>5.38</td>
<td>2.25</td>
<td>1.07</td>
</tr>
<tr>
<td>Squish</td>
<td>1.32</td>
<td>5.69</td>
<td>2.58</td>
<td>1.08</td>
</tr>
<tr>
<td>DeepSqueeze (CPU)</td>
<td>1.34</td>
<td>5.77</td>
<td>2.60</td>
<td>1.09</td>
</tr>
<tr>
<td>DeepSqueeze (GPU)</td>
<td>1.33</td>
<td>5.79</td>
<td>2.59</td>
<td>1.09</td>
</tr>
</tbody>
</table>

For Corel (Figure 5b), DeepSqueeze uses around 25% less space compared to Squish with an error threshold of 0.1. In the same dataset, for an error threshold of 0.005, the compression ratios achieved by the two methods are much closer.

The results for the Forest dataset (Figure 5c) show an even better improvement over Squish. The relative savings in space are about 5%, 6%, 40%, and 70% for the four different error thresholds (0.005, 0.01, 0.05, and 0.1), respectively.

For the three remaining datasets, the space savings achieved by DeepSqueeze are also significant. In the Census dataset (Figure 5d), we see that DeepSqueeze is able to offer a 28% improvement. More importantly, for the Perfmon dataset which has only numerical attributes, the compression ratio achieved by DeepSqueeze beats that of Squish by around 35% for \(e = 0.005\) and 68% for \(e = 0.1\). Lastly, for the largest dataset (Criteo) which is around 1TB in size, we see that DeepSqueeze is still able to outperform Squish by up to almost 33% which shows that our techniques can scale to significantly larger datasets.

The Census and PerfMon datasets are particularly interesting because the Census dataset is highly dimensional with low sparsity (as opposed to Forest cover dataset which is high dimensional but with high sparsity) and the Perfmon dataset is both large and contains all numerical attributes that are more difficult to predict accurately. Therefore, the superiority of DeepSqueeze in these two datasets shows that DeepSqueeze is able to discover more complex relationships between the attributes and uses a more efficient materialization method to achieve compression.

7.3 Runtime

In this section, we compare the running times for both compression and decompression operations using gzip, Parquet, Squish, and DeepSqueeze. Since DeepSqueeze is based on deep models, we show results with and without a GPU used to accelerate the model training and inference.

Table 1 shows the runtimes for both compression (left) and decompression (right) for four of the datasets after determining appropriate hyperparameters. On the second largest dataset, hyperparameter tuning in DeepSqueeze takes up to 30 minutes, whereas Squish takes several hours while still requiring manual tuning. For the largest dataset, DeepSqueeze hyperparameter optimization takes nearly 24 hours to complete. For Corel and Forest the whole datasets are used for training while for Perfmon and Criteo sample sizes of 10% and 1% are used respectively.

As shown, DeepSqueeze is able to offer reasonable performance for both compression and decompression. More specifically, both DeepSqueeze and Squish take longer than the other approaches since the model training step is computationally intensive and potentially involves multiple passes.
Finally, over the Criteo dataset, both DeepSqueeze as well as Squish take about 48 hours to compress the 1TB file on the tested hardware. However, depending on the application, the benefit in terms of space savings can sometimes be more important than the time required to compress the data (e.g., data archival). Importantly, compression in DeepSqueeze is embarrassingly parallel, allowing many nodes to simultaneously compress data (e.g., edge devices).

7.4 Microbenchmarks

This section includes experiments that evaluate the behavior of DeepSqueeze under different settings. First, we present results that show how DeepSqueeze compares to a simple baseline as well as the impact of our proposed optimizations. Then, we justify our choices of failure encoding and illustrate the effectiveness of Bayesian hyperparameter tuning.

7.4.1 Baseline Comparison and Breakdown. In order to measure the overall impact of our proposed techniques as well as compare our approach to a baseline, we show the benefit of each optimization for all tested datasets in Figure 8. More specifically, we show how (1) the complexity of the model, (2) quantization of numerical attributes, and (3) mixture of experts impacts the compression ratio. Figure 8.

As shown, a model with just a single hidden layer and linear activation function performs poorly across the board. Similarly, without quantization, the compressed output is considerably higher in all datasets except Census since it has no numerical attributes. Finally, we see that using multiple experts helps for the larger datasets, while the impact is not noticeable on the smaller Corel and Forest datasets. Criteo is affected the most among the three since of it is the largest and most complex, requiring more experts to learn the relationships between attributes.

7.4.2 Mixture of Experts. As explained in Section 5, clustering algorithms are straightforward way of partitioning the data in order to build specialized individual models. Most of these algorithms, however, cluster the given points based on closeness criterion rather than statistical/distributional properties. On the other hand, our mixture of experts approach learns to partition the dataset during the training process.

Figure 6 illustrates a comparison between the best compression ratio that was achieved using similar number of clusters (or experts) for a simple clustering algorithm (K-means) versus DeepSqueeze’s mixture of experts. The dataset that the experiments have been performed on is Perfmon.
For the lowest error threshold of 0.1 (Figure 6d), the lowest compression ratio without using any partitions is around 3.8%. As the number of experts/clusters grows to 4, the compression ratio gradually goes down to around 3.1% (i.e., around 18% improvement) for multiple experts and conversely it grows higher than 3.8% for K-means. This is because, adding each submodel introduces some storage overhead in the form of materialized deep models and orders files that guarantee the order of decompressed rows will be preserved. Similarly, for other error thresholds when the number of experts is 4, DeepSqueeze achieves the best compression ratio: for \( e = 0.05 \), the compression ratio starts from 6% and goes down to 5.7% (i.e., 5% relative improvement), for \( e = 0.01 \), it starts from 7.8% and goes below 6.5% (i.e., 16% relative improvement), and for \( e = 0.005 \), it starts from 8.9% and shrinks to 8.3% (i.e., 6.7% relative improvement).

### 7.4.3 Training Sample Size
For large datasets, DeepSqueeze can use a sample of the data to train the models in order to reduce the overall runtime of the compression task. In many cases, training over a sample of the dataset can offer large performance advantages with minimal loss in generalization. However, models that are trained using a small or non-representative sample might not generalize well to the entire dataset and therefore result in poor performance.

Therefore, in Figure 9, we show the compression ratio for various sample sizes of the PerfMon dataset with an error threshold of 10%. As shown, models trained using sample sizes of less than \( \approx 10\% \) are not robust enough to generalize, leading to poor compression ratios. On the other hand, models built using larger sample sizes offer little added benefit but take significantly longer to train.

### 7.4.4 Hyperparameter Tuning
As explained in Section 5, DeepSqueeze’s compression model has three main hyperparameters: (1) learning rate, (2) code size, and (3) number of experts. We allow the search algorithm to explore different learning rates, representation sizes, and number of experts in the range of \((10.0^{-5}, 10.0^{-4})\), \([1, \text{number of attributes}]\), and \([1, 10]\), respectively.

### 8 CONCLUSION
This paper presented DeepSqueeze, a deep semantic compression framework for tabular data. Unlike traditional semantic compression algorithms, DeepSqueeze can capture complex relationships in the underlying data by leveraging a type of artificial neural network called an autoencoder. DeepSqueeze supports lossy compression with error bound guarantees for enhanced data compression. Additionally, our compression techniques are “columnar-compression aware,” such that the outputs of our models can be further compressed using well-known columnar compression techniques. Overall, we observed up to almost \(4\times\) reduction in compressed output size compared to state-of-the-art alternatives.
REFERENCES

[34] L Peter Deutsch. 1996. DEFLATE compressed data format specification version 1.3. (1996).
[38] L Peter Deutsch. 1996. DEFLATE compressed data format specification version 1.3. (1996).
A Revision Letter

Dear SIGMOD Chair and Referees:

We would like to thank you for the helpful feedback concerning our submission. We have significantly improved the paper and believe we were able to address all revision requests, as well as many additional points made in the reviewer comments.

To summarize, we made the following major changes:

- Significant new experiments in Section 7, including a new large ≈1TB dataset (Criteo) and a comparison to a baseline autoencoder/breakdown of the benefits of our proposed optimizations.
- Clarification of our contributions and claims regarding existing work (Sections 1-3), including an expanded discussion of related work in Section 2.
- Better explanation and justification of our proposed optimizations, including a new set of experiments (Section 7.4.1 and Figure 8).

In the following, we summarize the comments from individual reviewers and how we addressed them in our revision. Our answers to specific points, as well as changes to the original submission, are highlighted in blue.

A.1 Reviewer #1

W1/D1/R1a/R1b: Clarify claims reg. autoencoders for compression + comparison with a baseline autoencoder.

As we noted in Section 2.4, autoencoders have been widely used for compression (primarily for images), as well as in other applications (e.g., denoising). However, unlike these works, we specifically target archival compression of tabular/relational data. Importantly, our proposed techniques can handle the mixed numerical/categorical attributes that characterize these datasets while respecting per-attribute error thresholds, unlike image data that is purely numerical. We have updated Section 2.4 to clarify these claims; also, please see our answer to W2/D2/R2 below.

Furthermore, we propose several novel optimizations that improve upon the size reductions achieved by a baseline autoencoder, and our new experiments (Section 7.4.1) show the improvement offered by each optimization.

W2/D2/R2: Clarify the handling of categorical and how they help in mixed distributions.

Yes, DeepSqueeze does learn relationships among both numerical and categorical attributes, as demonstrated by the four datasets that include a mix of both attribute types (Corel, Forest, PerfMon, Criteo). DeepSqueeze can also handle categorical-only datasets, as shown by the Census dataset. The intuition behind this is that, just as a numerical attribute can correlate with other numerical attributes, so too can categorical attributes be used to predict the values of other attributes. We have updated the text to better explain this.

D3a: Criteo

As requested, we have added several experiments with the Criteo dataset to Section 7. The headline result (i.e., compression ratio) is shown in Figure 5f, where DeepSqueeze achieves up to ≈33% size reduction compared to the state-of-the-art (Squish). We believe that these results demonstrate that our techniques can scale to large datasets that have complex relationships among attributes.

D3b/D3c: More controlled datasets (i.e., synthetic microbenchmarks)

This is a great suggestion, but we unfortunately could not complete these experiments before the 1-month revision deadline, although we would definitely plan to include them in a camera-ready. However, we believe that the real-world datasets already showcase DeepSqueeze’s efficacy in a wide variety of scenarios, including mixed numerical/categorical (Forest, PerfMon, Criteo), numeric-only (Corel), and categorical-only (Census), as well as varying dataset sizes, value distributions, and complex relationships among attributes.

D4: Do the runtimes of compression in Table 1 include all hyperparameter tuning?

As in related works, we do not include hyperparameter tuning in the reported runtimes in Table ???. Our approach is automated and takes around 30 minutes for PerfMon and around 24 hours in total for the 1TB Criteo dataset, whereas Squish takes many more hours than DeepSqueeze and requires manual tuning. We have updated Section 7.3 to better clarify this and to discuss how long hyperparameter tuning takes for each dataset.

D5: You also mention sampling (Sec 7.4.2) suffices for your model. Was this done for Table 1? If so, what were the percentages?

Yes, we did use sampling for some of the larger datasets. Specifically, we used a 10% sample for PerfMon and 1% for Criteo. Corel, Forest, and Census are sufficiently small that we can train on the full dataset with no sampling. We have updated Section 7.3 to specify these sample sizes.

A.2 Reviewer #2

W1. Comparison to other methods seems limited.

Our primary comparison point is Squish [18], which is the current state-of-the-art semantic compression algorithm. Since Squish was shown to outperform prior semantic compression algorithms (e.g., Spartan [11], ItCompress [25]), we did not include those approaches as comparison points.

Additionally, to give readers an idea of how DeepSqueeze compares to more traditional techniques, we show results for: (1) gzip [15] and (2) Parquet [29]. However, these methods cannot match the small compression sizes of the semantic compression algorithms (i.e., Squish and DeepSqueeze), as shown in Section 7.2 and Figure 5.
Moreover, as part of this revision, we have added experimental results that compare DeepSqueeze to a baseline autoencoder with none of our optimizations. The results (Section 7.4.1 and Figure 8) measure the impact of our proposed optimizations and show that our techniques allow us to achieve significantly better compression than the baseline.

W2. Straightforward application of autoencoders to the problem (e.g., how large-cardinality columns or varying error thresholds affect efficacy/compression ratios).

As explained in our answer to Reviewer #1 W1/D1/R1a/R1b, we clarified the claims regarding our specific contributions (Sections 1 and 2). While we note that autoencoders have been previously used for compression, Sections 4-6 describe several additional techniques and optimizations that we propose specifically for tabular/relation data. As previously mentioned, we have added a new experiment that compares DeepSqueeze to a baseline autoencoder and shows the improvements offered by each individual technique.

For high-cardinality attributes (e.g., unique values or primary keys/IDs), you are correct that our techniques will not work because there is no relationship or pattern to learn with other columns. In such cases, we can automatically detect this and fall back to standard compression techniques. We have updated the text in Section 4 to better explain how we handle these cases.

For high-cardinality attributes (e.g., unique values or primary keys/IDs), you are correct that our techniques will not work because there is no relationship or pattern to learn with other columns. In such cases, we can automatically detect this and fall back to standard compression techniques. We have updated the text in Section 4 to better explain how we handle these cases.

Finally, as you point out, the choice of error threshold for numeric attributes can have a substantial impact on compressed file size. For example, going from a 0.5% to a 10% error threshold on the Forest dataset (Figure 5c) results in a ~8 X size reduction in DeepSqueeze’s compressed output, whereas the difference for Squish is only ≈20%. The error thresholds we tested (i.e., 0.5%, 1%, 5%, and 10%) represent a broad range of tolerance for lossiness and are standard values used in other semantic compression publications (e.g., Spartan [11], ItCompress [25], Squish [18]). Therefore, we believe that the experiments in Section 7 demonstrate how choice of error threshold impacts DeepSqueeze’s performance, especially with the addition of the large Criteo dataset (Figure 5f) and the baseline comparison/breakdown (Figure 8).

Related work: Huge amount of literature in data mining on compressing tabular data (MDL); e.g. see Jilles Vreeken’s thesis

Thank you for pointing this out. We have added it to our discussion of related work in Section 2.

The quantization of numerical attributes makes sense for input but for output - how is overlapping nature dealt with? E.g. if the correct age 59 and the tolerated error is 2 years, then using buckets 56-57, 58-59, 60-61 would all be sufficiently correct.

You are correct that overlapping buckets (as in your example) would indeed yield multiple possible model outputs. However, the quantization phase of preprocessing (Section 4.2) produces non-overlapping buckets identified by their midpoint as the input to the model, and the model can produce only these buckets as output. Since each value can therefore map to exactly one of the non-overlapping buckets, the model must output the same bucket given as input during decompression. Otherwise, the model output is marked as a failure and stored accordingly (Section 6.3).

As an example, consider values in the range 0 to 100 with 10% tolerated error. The quantized buckets are each non-overlapping intervals of 20 values, which are represented by: 10, 30, 50, 70, and 90. Your example value of 59 could therefore only map to bucket 50 while respecting the 10% error threshold. In this case, though, a value like 60 could be correctly represented by both the 50 and 70 bucket, but this problem can be solved by making all but the first or last bucket either left- or right-exclusive, respectively (e.g., [40, 60) and (60 – 80), etc.).

As part of this revision, we have updated Sections 4.2 and 6.3 to better explain our quantization strategy.

For binary values, isn’t storing correct/incorrect just as much information as storing the correct value?

You are correct that the same single bit is required to store the original 0/1 value as our failure encoding scheme (0 for correct, 1 for incorrect). However, the advantage of our approach comes when the model makes many correct (or systematically incorrect) predictions that transform the original data into long runs of either 0 or 1, which can then be effectively compressed using techniques like run-length encoding.

For example, consider a binary attribute with 5 values: [0, 1, 0, 1, 0]. If we correctly predict the first 4 values and mispredict the last, the materialized output will be: [0, 0, 0, 0, 1]. We can then run-length encode the long run of 0s.

A final important note is that we needed to adapt the original categorical algorithm of storing a null/sentinel value when the model is correct, or the original value in the event it is wrong (i.e., 0 or 1), since that would require a total of 3 values (or 2 bits). By storing a 0 for correct and a 1 for incorrect (i.e., flip the prediction of the model), we still require only 1 bit per value.

We have updated Section 6.3 to better explain this.

For non binary values, why not use the probabilities produced by the model to compress further in cases where the model is wrong?

Thank you, this is an excellent idea. The algorithm would work something like the following.

Assume a categorical attribute with possible values \{A, B, C\}. For a particular data item, suppose our model produces a probability distribution \{A = 0.7, B = 0.1, C = 0.2\}. Currently, we would predict the value with the highest probability, in this case A. If that prediction is correct, we would
store null/0; otherwise, we would store the correct value (e.g., C/2).

However, if we instead sorted the predictions by decreasing probability, we could store the index of the prediction that matches the ground truth. In the example, the value C would thus be stored as 1 instead of 2. Assuming the first few predictions of the model are often correct, a variable-length compression scheme (e.g., Huffman coding) could significantly reduce the size.

Unfortunately, we could not implement and test this optimization within the 1-month revision deadline, but we plan to try it as an immediate next step and hopefully include the results in a camera-ready.

MoE with different size experts.

This is a great idea for an additional optimization that could reduce (1) the model size for the materialized decoder, leading to a smaller compression ratio; and (2) the overall runtimes for both the training/compression and decompression phases, since simpler models would be faster. Currently, DeepSqueeze is pessimistic in that all experts are as complex as the most complex expert required, but we could potentially do much better if some experts could be much simpler. However, regularization in the representation layer mimics the idea of using different expert sizes because some dimensions of the representation may end up being all zeros and therefore be dropped by the columnar compressor. Again, although we could not explore this optimization in the 1-month revision deadline, we definitely plan to look into this idea as future work.

A.3 Reviewer #3

D1: Unclear Use Case / Motivation (W1)

As briefly mentioned, we focus on use cases like long-term archival (e.g., scientific datasets) and bandwidth-limited transmission (e.g., sending telemetry data from autonomous vehicles/mobile phones to the cloud for offline learning) that can tolerate longer compression times in exchange for significant size reduction, since latency is not critical. However, making compressed data queryable introduces an interesting trade-off between size and latency, which we are currently exploring in a follow-up paper. We have updated the introduction/motivation to better reflect this.

D2: Compression Overhead (W2)

While it is true that our compression time is higher than traditional approaches like gzip [15] or Parquet [29], DeepSqueeze can achieve much smaller compression sizes, effectively trading off more processing time for a reduction in size. Moreover, our results also show that DeepSqueeze is significantly faster than Squish [18], the current state-of-the-art semantic compression algorithm, while simultaneously achieving significant size reductions.

These higher compression times are often tolerable in large-scale data archival where latency is not critical. For further discussion of use cases and associated trade-offs, please see our answer to W1/D1.

D3: Unclear Experiments (W2)

We have clarified many of these points by updating the text in Section 7.

Specifically, the definitions, datasets, and experimental setup that we use are standard conventions used in prior semantic compression publications (e.g., Spartan [11], ItCompress [25], Squish [18]). In these papers, the baseline is the size of the uncompressed file, and the “compression ratio” achieved by an algorithm is calculated as a percentage of this uncompressed file (from Squish Section 6.1):

\[
\text{compression ratio} = \frac{\text{data size with compression}}{\text{data size without compression}}
\]

Similarly, the tested error thresholds for numerical attributes of 0.5%, 1%, 5%, and 10% are standard values used in the other semantic compression publications. Error thresholds of 0% (i.e., lossless compression) are generally not possible for semantic compression algorithms, since reconstructing arbitrary-precision numbers devolves into reconstructing unique values (please see our answer to Reviewer #2 W2). Since the Census dataset has only categorical attributes, which must be losslessly reconstructed, it is the only one to have an error threshold of 0%.

You are correct that reducing the number of epochs while training over more of the dataset would produce the same (or better) results. However, from a practical implementation perspective, it is much more efficient for very large datasets (e.g., Criteo is 1TB) to run more epochs on a sample that can fit in memory than to stream over the full dataset from disk/SSD. For small datasets (e.g., Corel, Forest, Census), sampling is not necessary, since they can fit entirely in memory. We have updated the text to better explain when/why we perform sampling.

Finally, regarding training/hyperparameter tuning times, we have updated Section 7.3 to better clarify our results. Please also see our answer to Reviewer #1 D4.

D4: Unclear Handling of Categorical Data (W3)

Please see our answer to Reviewer #1 W2/D2/R2.

D5: No Query Processing (W3)

Please see our answer to D1/W1 above.

D6: Related Work

Thank you for bringing these related works to our attention. We have added them to our discussion in Section 2. We have also updated the introduction to better clarify our specific contributions.

D7: Presentation

Thank you for pointing these out. We have corrected the mentioned issues with typos/wording, as well as updated the text to clarify the descriptions of our approach.