Neural Data Compression

Floris Kornelis van Dijken Wout Verdyck

November 2024

Abstract

This paper explores the potential of neural data compression for textual and binary data, using deep learning models to develop more adaptive and efficient methods for compressing data. Our study provides a comparison between traditional compression techniques and the proposed neural compression methods, focusing on key metrics such as compression ratio and execution time. We analyze the performance of these models, providing insight into their practical applications and potential for future development.

1 Introduction

Data compression involves the process of encoding information to use fewer bits than its original format [1]. By reducing the size of the data, we can achieve faster data transmission across networks and require less storage space. While lossless neural network-based compression has been successfully implemented for various data types, such as images [2] [3] and audio [4], this work focuses on the compression of textual data, including ASCII text, DNA sequences, and binary executables. There are several methods for lossless compression of textual data, which can be categorized into distinct approaches.

Entropy encoding: This method reduces the bit length of each character based on its frequency of occurrence, performing well for data with a significant difference between character frequencies. Examples are Huffman coding and arithmetic coding.

Dictionary-Based Compression: This method works by identifying repeated patterns and replacing them with references to a dictionary. This approach is particularly effective for data with high internal correlation. Some examples are LZW, LZ78, LZMA, zlib, gzip, and brotli [5].

Neural network-based compression: This paper will focus on this method. A neural network is trained to predict the input and use entropy encoding to compress the transformed representation.

There are several more types of compression methods such as statistical data compression techniques or hybrid methods which will not be covered in this paper.

2 Related Work

Neural network-based approaches to data compression have gained attention in recent years, particularly for their ability to learn complex patterns and correlations in data. Transformer-based models are the state-of-the-art in predicting and generating text by using the self-attention mechanism to capture long-range dependencies. Recent works have adapted transformers such as GPT-2 for data compression tasks by predicting the next token in a sequence [6].

Another promising method is DZip [7], which uses Recurrent Neural Networks (RNNs) for text compression. DZip has demonstrated better compression performance compared to Gzip, 7zip, and BSC on specific datasets.

There are several methods for encoding predicted data, one of which involves making deterministic predictions to reduce redundancy. [8]. For each character, the model predicts what the next character could be. If the actual character matches the prediction, it is considered redundant and is ignored. In the cases where the prediction is incorrect, the actual character and the number of consecutive correct predictions since the last mismatch are recorded in a separate file. An escape character is used to distinguish between control data and actual text. The second file, which contains the mismatches and their associated data, is further compressed using Huffman coding, producing the final compressed file. The paper demonstrates that while this approach reduces the input size, there are more effective methods for encoding the predicted data.

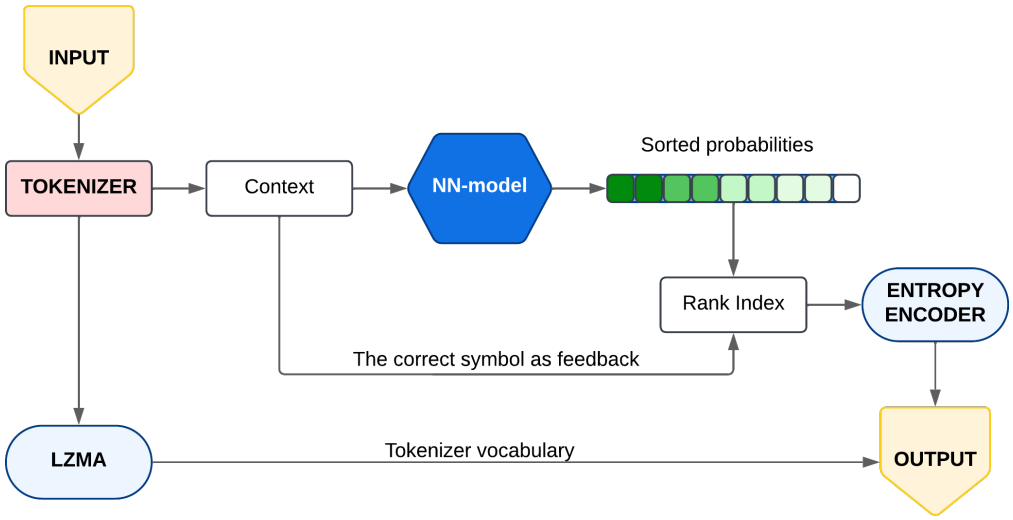


Figure 1: Diagram representing the compression process

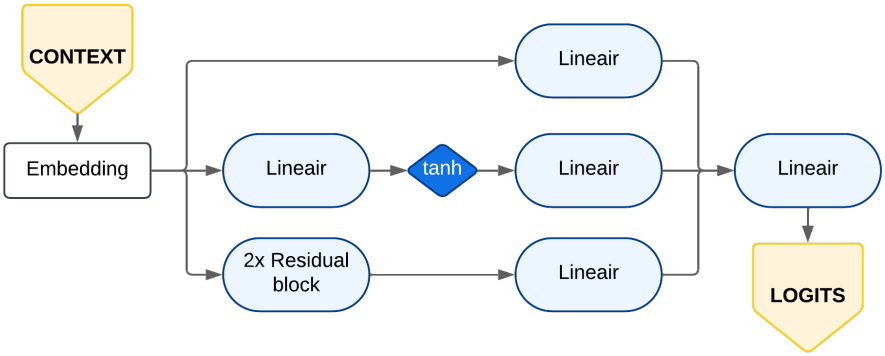


Figure 2: Diagram representing the architecture of neural network used in the supporter model

3 Methods

3.1 Model Rank Generation

Given a string S composed of characters $\{s_0, s_1, \ldots, s_{n-1}\}$ from a finite alphabet A, the model's task is to predict the next character s_i. The prediction is made based on the preceding substring of length k, conditioned as $P(s_i \mid s_{i-1}, s_{i-2}, \ldots, s_{i-k})$, where P represents the conditional probability distribution.

Once the probability distribution for each character s_i in the alphabet is calculated, the model's performance can be assessed by comparing its predictions to the actual characters in S. Specifically, we count how many characters the model ranked as more probable than the actual character s_i from S. This process results in a transformed representation where each element corresponds to a rank value between 0 and $|A| - 1$.

This transformation skews the distribution of rank values, with a higher frequency of smaller values - for example, more 0's than 1's, more 1's than 2's, and so on. This skewed representation is then fed into an arithmetic encoder, which reduces the overall data size by efficiently encoding the distribution.

3.2 Model Architectures

3.2.1 Static model

The static model uses a fixed vocabulary - comprising of mostly ASCII characters for English text, and the bases A, C, G and T for DNA sequences. The model processes a series of input characters through the following layers:

- *Embedding layer*: This layer converts characters into dense vectors of a fixed size. These are more suitable for neural network processing.
- *Long Short-Term Memory (LSTM) layer*: This layer captures temporal dependencies and learns patterns in the data.

- *2x Dense layer*: This fully connected layer, with ReLU activation, introduces non-linearity, allowing the model to learn more complex patterns in the data.

As a result, the model produces a probability distribution from which the most probable, second most probable, and so on, characters can be determined.

3.2.2 Supporter Model

The Supporter Model shown in figure 2 predicts the next character in a sequence by using neural network architectures designed for efficient sequence modeling. This model combines three distinct networks to capture both simple and complex features in the input.

Tokenizer
Tokenization involves breaking down input text into smaller chunks that can be processed by the model. We will test methods both with and without tokenization. In the approach without a tokenizer, each character in the text is treated as a separate symbol. For text using ASCII encoding, there are at most 128 different symbols.

In the tokenized method, the vocabulary is initialized by treating each character as a unique token. The algorithm then iteratively merges the most frequently occurring pairs of symbols into new symbols, which are added to the vocabulary. This process continues until the desired vocabulary size is reached. In our experiments, the vocabulary size is adaptively chosen based on the input size, ranging from 150 to 1500 tokens. The mapping between symbols and their corresponding tokens is stored in the output file, which is further compressed using *LZMA* to reduce overhead. Balancing the vocabulary size is important, as larger vocabularies can increase computational overhead during model training and rank computation. The tokenizer can be seen in figure 1.

Architecture
The model takes the tokenized vector as input, where each integer token is transformed into a dense vector embedding. These embeddings are then processed through three neural networks:

Linear Network: This component applies a fully connected layer directly to the embeddings, serving as a baseline for capturing simple relationships in the sequence.

Dense Network: This network uses a two-layer feed-forward neural network with a `tanh` activation function. The non-linear activation enables it to model complex non-linear dependencies between the input tokens.

Residual Network: This network consists of two linear residual blocks, each composed of two fully connected layers with `ReLU` activations.

The outputs from the linear, dense and residual networks are concatenated to form a combined feature representation, which is then passed through a final fully connected layer to produce the logits. The logits are subsequently passed through a softmax function to compute the probability distribution over the alphabet A.

To reduce the model size, we optionally apply quantization to the model's inputs and outputs. This process converts floating-point tensors into lower-precision representations, reducing the model size without significantly degrading performance.

Recurrent Neural Network Variation
The network shown in 2 exclusively uses fully connected layers, which are efficient but limit the model's ability to capture complex features, such as sequential dependencies in text. To address this limitation, an alternative version replaces the two residual blocks with a single Recurrent Neural Network (RNN) block [9]. RNNs are better at modeling patterns and dependencies.

The RNN unit maintains an internal hidden state, which is updated at each step based on the current input and the previous hidden state, allowing the network to capture more dependencies in the data. This feedback loop lets the network learn from past inputs and use that knowledge as it processes new information.

While the RNN block improves the model's ability to capture sequential patterns, it also increases computational complexity, leading to slower learning compared to the fully connected variant.

3.3 Compression Methods

3.3.1 Static method

In the static method, we use a pre-trained model which is trained on a large amount of similar data. This pre-trained model is shared by both the compressor and decompressor, so there is no need to include it to the output. To ensure good performance across various datasets, overfitting should be minimized to maintain accuracy.

A major drawback of the static method is that the trained model will only perform well on data with a similar structure. For example, a static model trained on DNA sequences will perform poorly when applied to English text.

3.3.2 Dynamic Method

In the dynamic method, we do not use a pre-trained model. Instead, the model is trained directly on the input data. The Supporter model is trained for 40 epochs on the input before encoding. The advantage of this approach is that the model only needs to perform well on the given input sequence, meaning generalization is not required.

A drawback of this method is that the decompressor needs access to the exact same model used by the com-

pressor. As a result, the model and its parameters must be included in the output, which introduces overhead, particularly for smaller input sizes. This creates a trade-off between model size and compression efficiency.

3.3.3 Adaptive method

In the adaptive method, the model size overhead is eliminated by training the model while compressing the data. During decompression, the inverse process is applied, where the model is trained on the already decompressed data. A key drawback of this approach is that the model is less effective at the beginning of the input due to the lack of prior information.

The adaptive method learns character by character, but this can be slow. To address this issue, a batch variant is introduced, where the input is divided into chunks and the model learns on a chunk-by-chunk basis. This method also allows for compressing data streams. This optimization cannot be applied during decompression because we need the previous n characters to predict the next one. If the character $n-1$ is not yet known, a prediction cannot be made.

3.4 Datasets

We use a diverse set of real-world datasets to evaluate our methods:

- *text8*: The first 4MB of English text extracted from the enwiki9 dataset, a subset of the English Wikipedia dump.
- *bible*: The complete English version of the Bible, offering natural language and grammatically correct sentences.
- *webster*: An HTML page from the 1913 Webster Dictionary, serving as a source of structured text data.
- *mozilla*: A binary file of the Mozilla 1.0 executable, representing a typical software binary dataset.
- *chr20*: Chromosome 20 from the Homo sapiens GRCh38 reference sequence, used for testing the methods on genomic data.
- *celegchr*: A dataset of *Caenorhabditis elegans* DNA sequences, offering another example of genomic data for evaluation.

3.5 Hyperparameters

In the experiments, we used the following hyperparameters for the Static method:

- *For English text*: A hidden size of 128, a learning rate of 0.001, a batch size of 256, and a sequence length of 30. The model ran for 6 epochs, with early stopping patience set to 10.
- *For DNA sequences*: A hidden size of 128, a learning rate of 0.01, a batch size of 128, and a sequence length of 50. The model ran for 20 epochs, with early stopping patience set to 5.

For model training and validation with English text, we trained the model on the *bible* dataset. Approximately 1 MB of the dataset was reserved for testing, while the remainder was split into 90% for the training set and 10% for the validation set, which was used for hyperparameter tuning. We did the same for the *chr20* dataset.

For the Adaptive and Dynamic methods, we used the following hyperparameters:

- *Adaptive method*: A hidden size of 64, an initial learning rate of 0.001, a minimum learning rate of 0.00005, and a decay rate of 0.9999. The batch size was 128, and the vocabulary size was 128, except for binary datasets, where the vocabulary size was set to 256.
- *Dynamic method*: A hidden size of 58, a learning rate of 0.01, and the model was trained for 40 epochs.

These hyperparameters were chosen based on preliminary experiments to strike a good balance between model performance and computational efficiency.

For the tokenized version, the vocabulary size was set to 1500 for the tokenized adaptive method and 1000 for the tokenized DynamicCompressor.

4 Results

4.1 Rank Prediction

If the model performs well, we expect a higher concentration of lower ranks. The histogram in Figure 3 shows that 80% of the predictions fall within the first 5 ranks. A greater concentration of low ranks indicates a more skewed probability distribution, leading to a lower entropy representation and better achievable compression rates. It's key to emphasize that the generated ranks themselves do not compress the input; the list of ranks simply provides a different form of input with a higher compressibility through techniques such as entropy encoding.

Figure 5 illustrates the behavior of the adaptive method when compressing the *bible* dataset. Initially, the entropy is high. After processing approximately 10,000 characters, the entropy drops below that of the original file. As the input size increases, the entropy continues

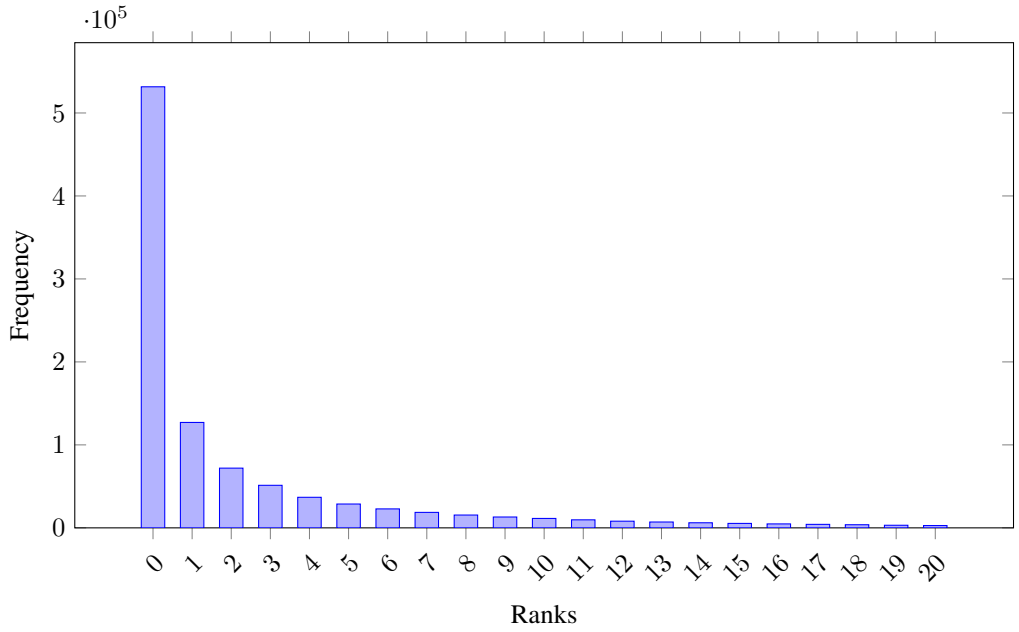


Figure 3: Frequency of characters after running adaptive method with RNN on the first 1MB of the *bible* dataset

to gradually decrease. Even after processing 1 million characters, we observe a persistent downward trend in entropy, suggesting that the model is still learning and refining its predictions.

4.2 Compression Performance

Table 4.3 shows the compression performance for each method expressed in bits per character (bpc). The lower the bits per character, the better the compression method.

$$bpc = \frac{compressed\ size\ in\ bits}{input\ size\ in\ bits}$$

When comparing neural compression methods with arithmetic encoding, it is clear that the proposed models achieve better compression rates, with one exception being the static method on the *webster* dataset. This can be explained by the frequent use of special characters that are never used in the *bible* training set on which the static model was trained.

For tokenized methods, we observe an improvement of 40% to 50% on textual datasets, while non-tokenized methods show a moderate gain of approximately 25% to 30%. For the mozilla executable, the improvement ranges between 10% and 20%, indicating that the executable data contains fewer patterns compared to English text. For the DNA datasets, the compression rates are nearly identical, suggesting that the model struggled to detect patterns within the DNA sequences.

When comparing RNN-based methods to their non-RNN counterparts, tokenized versions show a minor decrease in bits per character, typically below 10%. In contrast, non-tokenized RNN methods show a more significant decrease in bits per character. For instance, the Adaptive RNN method shows an improvement of 20% to 25% on datasets such as *webster* and *bible*, while the Dynamic RNN achieves approximately 15% better compression rates on these datasets. The results for binary and DNA sequences remain consistent across both approaches.

Comparing tokenized methods to their non-tokenized counterparts, it is evident that tokenization consistently improves compression performance across all datasets. Tokenized versions of non-RNN methods, on average, achieve a reduction of 1 bit per character (bpc) compared to the non-tokenized methods, corresponding to an approximate 40% improvement in compression ratio. Notably, the tokenizer was unable to efficiently tokenize the DNA datasets and these datasets were therefore excluded from the tokenized results.

Finally, neural compression methods are benchmarked against standard compression algorithms. All neural methods achieve a lower bpc than Huffman coding and outperform gzip and zlib on most datasets - except the aforementioned static method on the *webster* dataset. However, when compared to advanced dictionary-based techniques like Brotli and bz2, the neural methods perform worse. This suggests that traditional dictionary-based approaches remain highly effective, especially for structured or highly compressible data such as text.

4.3 Benchmarks

An essential aspect of compression is the time required for both compression and decompression. Figure 4 presents the runtime of each method applied to the first 1 MB of the *bible* dataset. For data compression, the dynamic method achieves a 3x speedup compared to the adaptive method. When comparing compression and decompression times, we observe a twofold increase in runtime for the adaptive methods and up to a sixfold increase for the dynamic method. This slower decompression performance is due to the limited optimizations available. While the compressor can use parallel processing for each batch of input, the decompressor requires sequential processing due to the autoregressive nature of the model. In autoregressive models, the prediction of a character s_i depends on previous characters $s_{i-1}, s_{i-2}, ..., s_{i-\text{context size}}$, which prevents parallel execution in the decompressor. It is important to note that the static method does not include data points for decompression, compression with RNN, or decompression with RNN in Figure 4. We did not fully optimize the static compressor and as a result, the compression and decompression speeds were not up to par. To avoid skewing the graph, we exclude the decompression time from the comparison. Additionally, the static compressor does not have an RNN variant, so that aspect could not be measured. Focusing on the RNN variant, the dynamic method shows an increase in runtime, primarily due to the computational cost of training RNNs on large input sizes. Approximately 90% of the runtime is spent on processing the first 150,000 characters (or 75,000 characters for RNN-based methods). Thus, for larger inputs, such as 2 MB, the runtime does not scale linearly and will be less than double. Decompression for the RNN-based dynamic method is only 1.2 times slower compared to the sixfold slowdown observed with the non-RNN version. Interestingly, for the adaptive method, there is no observable increase in runtime when using the RNN-based approach.

4.4 Model Variants

Model	Time (s)	Output size (bpc)
None	9.00	3.72
Residual	13.00	3.64
RNN 1 layer	13.61	3.42
RNN 2 layers	16.82	3.36
RNN 3 layers	19.74	3.42
LSTM	27.00	3,51
GRU	36.00	3.43
Transformer	112.35	3.67

Table 2: Comparison of different models based on execution time and model performance. Tested on the first 100KB of the *bible* dataset using the adaptive method.

While the benchmarks include two variations of the SupporterModel architecture — namely, the Recurrent Neural Network (RNN) and residual network architectures — we also experimented with several other model variations. Table 2 lists all the neural networks we tested. In these experiments, we replaced the residual block with the models shown in the first column of the table. For a baseline, the first row shows the performance without any model, resulting in the fastest execution time but the highest bits per character (bpc).

Examining the Recurrent Neural Networks, we observe an improvement of 0.22 bpc compared to the residual network, without a significant increase in execution time. We tested RNNs with additional layers. While two layers show a clear improvement, adding a third layer yields performance similar to a single layer, with each additional layer adding approximately 3 seconds to the execution time.

For LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), and Transformer models, the results indicate higher bpc and longer execution times compared to RNNs. This is likely due to their increased complexity, which may require more epochs to effectively capture patterns in the data.

5 Discussion

Although we experimented with several concepts, the evaluation was limited to relatively small input sizes (typically 1MB) due to the speed constraints of certain compression and decompression processes.

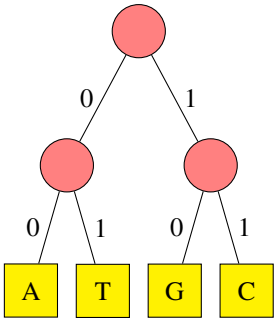
5.1 DNA Compression Limitations

Our results show that our neural network methods do not achieve high compression improvements over standard arithmetic encoding when applied to DNA sequences. To intuitively understand why our model performs poorly on DNA data, consider the use of Huffman encoding for rank-based compression. Although Huffman encoding typically yields slightly worse compression ratios than arithmetic encoding, both methods aim to minimize the number of bits per character.

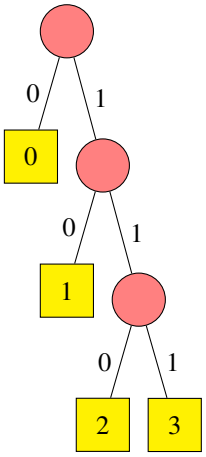
In DNA sequences, the vocabulary size is limited to four bases: A, C, G, and T. These bases generally have similar frequencies, which leads to a Huffman tree with uniform bit assignments:

Method	webster	bible	text8	mozilla	chr20	celegchr
Static	5.19	2.60	3.27	-	1.92	1.94
Dynamic	3.71	3.53	3.55	7.03	1.98	1.92
Dynamic RNN	3.16	3.13	3.3	7.19	1.97	1.92
TokenizedDynamic	2.72	2.12	2.41	6.08	-	-
TokenizedDynamic RNN	2.44	2.07	2.39	5.96	-	-
Adaptive	3.63	3.46	3.53	6.67	1.98	1.93
Adaptive RNN	2.72	2.74	3.08	6.48	1.96	1.93
TokenizedAdaptive	2.7	2.1	2.47	5.78	-	-
TokenizedAdaptive RNN	2.47	2.07	2.5	5.77	-	-
arithmetic	4.94	4.39	4.12	7.26	2	1.95
huffman	4.98	4.43	4.15	7.29	2.23	2
gzip	2.33	2.24	2.64	5.26	2.14	2.11
bz2	1.72	1.7	2.1	5.28	2.05	2.04
zlib	2.36	2.27	2.65	5.26	2.24	2.19
lzma	1.9	1.8	2.2	5.11	1.93	1.88
brotli	1.85	1.8	2.14	5.06	1.84	1.78

Table 1: Bits per character for each dataset, tested on the first 1MB of each dataset. To facilitate comparison of the methods, the model trained with the dynamic method is not included in the total compression output size. The size of this model is approximately 1.1MB for the tokenized version and 300KB for the non-tokenized version.



In this case, each character is encoded using 2 bits. For an input of 100,000 characters, the output size is 25,000 bytes. Ideally, our neural data compression method would produce a Huffman tree where the most frequent character is assigned a single bit:



Achieving this optimized Huffman tree requires the condition $freq(2)+freq(3) < freq(0)$, assuming $freq(3) \leq freq(2) \leq freq(1) \leq freq(0)$. However, when applying the dynamic method with an RNN to the *chr20* dataset (using the first 100,000 characters), we observed the following frequencies:

$$freq(0) = 308,440 \qquad freq(1) = 272,668$$
$$freq(2) = 249,420 \qquad freq(3) = 169,471$$

In contrast, the original DNA base frequencies were:

$$freq(A) = 271,844 \qquad freq(T) = 265,885$$
$$freq(C) = 234,169 \qquad freq(G) = 228,102$$

From these results, we observe that $freq(0) < freq(2) + freq(3)$, meaning that the Huffman tree generated after transformation remains the same as the one before the transformation. As a result, using Huffman coding on the transformed representation does not improve the final compression size.

This limitation explains why our neural compression method doesn't achieve better compression efficiency for DNA sequences: the differences in frequency distributions before and after processing are too small to yield improved compression.

6 Conclusion

This paper explored neural network-based approaches for lossless data compression across diverse data types, including text, binaries, and DNA sequences. Our results show that static, adaptive, and dynamic neural models—especially when combined with tokenization and

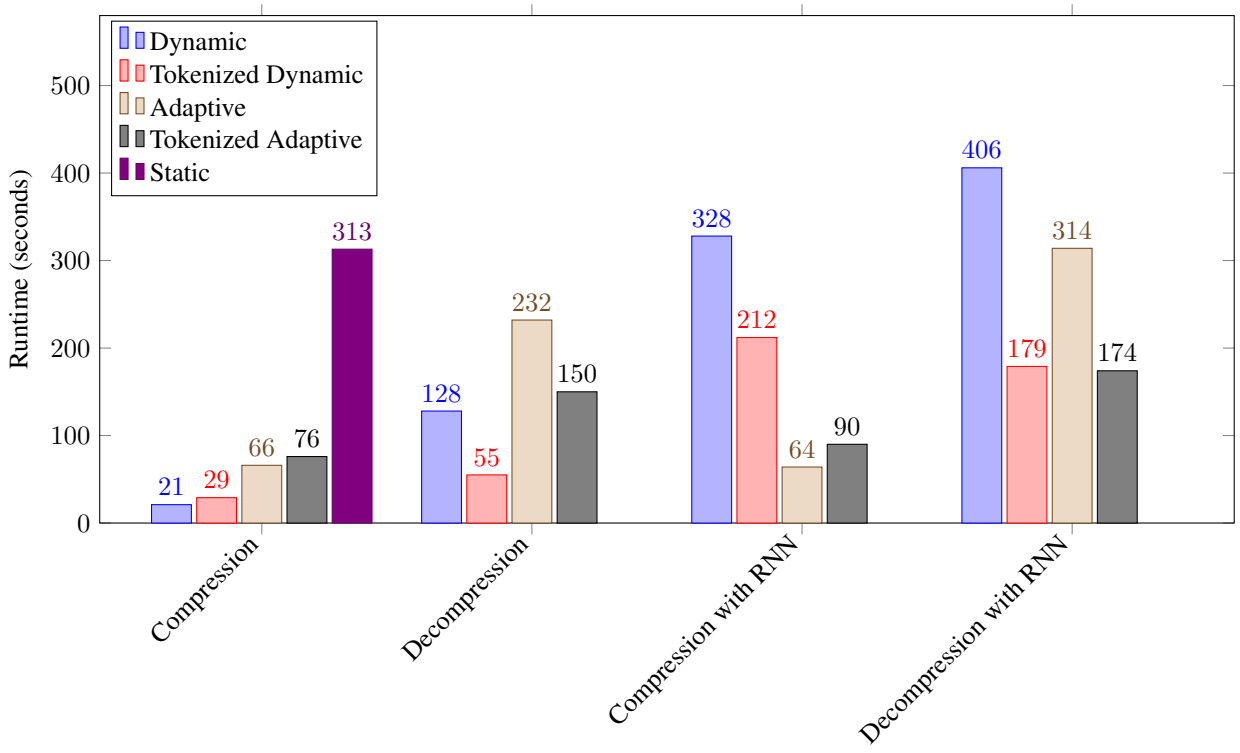


Figure 4: Runtime comparison of different methods on first 1MB of *bible* dataset. The benchmarks were run on an Apple M1 chip with an 8-core CPU and 8 GB of unified memory, with all tasks executed on the CPU.

recurrent layers—offer up to 50% better compression ratios compared to standard entropy encoding and also achieve higher compression ratios than conventional algorithms like Gzip and Zlib. However, these models don't surpass advanced dictionary-based techniques like Brotli, lzma and Bz2. We observed that the model's ability to compress DNA sequences was limited due to its difficulty in producing a more compressible output representation. The dynamic model has faster compression times but take longer to decompress. We also found that RNN-based models provided better compression ratios compared to fully connected methods, but at the cost of increased runtime.

7 Future work

We can improve existing methods by optimizing them further, such as using GPU capabilities and further tuning the model architectures. This GPU acceleration will enable the exploration of a wider range of model variants for dynamic methods, which were too computationally expensive to test.

7.1 Dictionary-Based Neural Compression

Our neural network transforms text into a representation with decreasing character frequencies, which improves the performance of arithmetic encoding. We found that dictionary-based methods, such as LZMA, are more effective for highly correlated inputs like English text. An interesting next step would be to convert input text into a format that is more easily compressed by dictionary-based algorithms. Discovering such a transformation could improve the compression ratio compared to rank-based neural data compression. For each state in the dictionary, we could calculate a list of the best compressible characters. This list would be sorted, starting with the character that requires the fewest bits in the output.

We thought of two strategies to achieve this transformation. The first approach is similar to rank-based compression where we calculate the rank and add it to the most compressible character. This method might be effective since the model often generates a 0 rank, in that case we add the best compressible character to the output. The second approach involves using multiple models in parallel, each producing a single output (such as the rank). We would then select the output that is most compressible according to the dictionary. This approach has a drawback: we must store which model was selected. These are

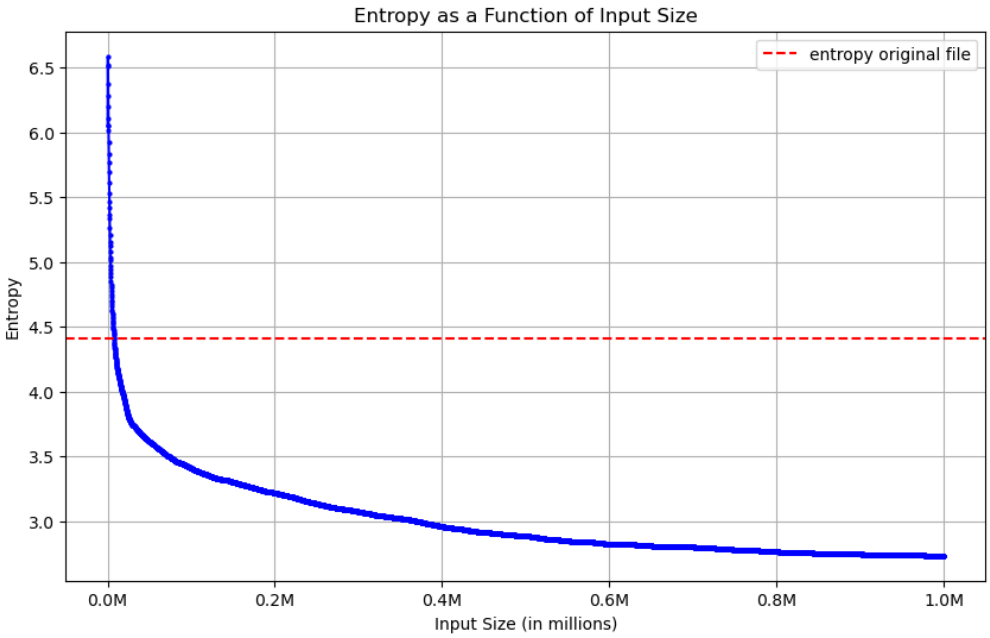


Figure 5: Entropy in function of the input size. The adaptive method with RNN is used on the first 1MB of the bible dataset.

initial ideas that could be refined or replaced with more efficient techniques.

7.2 Faster Decompression

A study analyzing Google's data centers found that, on average, each compressed byte was decompressed 3.3 times [10]. Given this data imbalance, improving systems throughput by developing and exploring techniques for faster decompression would be an interesting topic.

References

[1] *Data compression*. https://en.wikipedia.org/wiki/Data_compression. Accessed: 2024-11-26. n.d.

[2] Fabian Mentzer et al. "Practical Full Resolution Learned Lossless Image Compression". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[3] Fabian Mentzer, Luc Van Gool, and Michael Tschannen. "Learning Better Lossless Compression Using Lossy Compression". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[4] Henrique S. Malvar. "Lossless and Near-Lossless Audio Compression Using Integer-Reversible Modulated Lapped Transforms". In: *2007 Data Compression Conference (DCC'07)*. 2007, pp. 323–332. DOI: 10.1109/DCC.2007.51.

[5] Jyrki Alakuijala et al. "Brotli: A General-Purpose Data Compressor". In: *ACM Transactions on Information Systems* (2019). URL: https://dl.acm.org/citation.cfm?id=3231935.

[6] Swathi Shree Narashiman and Nitin Chandrachoodan. *AlphaZip: Neural Network-Enhanced Lossless Text Compression*. 2024. arXiv: 2409.15046 [cs.IT]. URL: https://arxiv.org/abs/2409.15046.

[7] Mohit Goyal et al. "DZip: improved general-purpose loss less compression based on novel neural network modeling". In: *2021 Data Compression Conference (DCC)*. 2021, pp. 153–162. DOI: 10.1109/DCC50243.2021.00023.

[8] J. Schmidhuber and S. Heil. "Sequential neural text compression". In: *IEEE Transactions on Neural Networks* 7.1 (1996), pp. 142–146. DOI: 10.1109/72.478398.

[9] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: 1912.05911 [cs.LG]. URL: https://arxiv.org/abs/1912.05911.

[10] Sagar Karandikar et al. "CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589074. URL: https://doi.org/10.1145/3579371.3589074.