Lossless Neural Text Compression

Stanford CS224N Custom Project

Michael Herrera
Department of Symbolic Systems
Stanford University
herrerx@stanford.edu

Kasey Luo
Department of Computer Science, Psychology
Stanford University
kaseyluo@stanford.edu

Abstract

Given the rapid generation of text-based information, the need for efficient compression is more important than ever. Advances in NLP present a huge opportunity to apply semantically-informed approaches to text compression. Our project seeks to push the frontier of text compression with a transformer-based neural network coupled with two data compression algorithms: variable-length integer encoding and arithmetic encoding. Our preliminary findings reveal that our neural text compression achieves 2X the compression ratio of the industry-standard Gzip. Though our neural-based compression takes significantly longer to compute compared to our baselines, we found through windowing that we can reduce computation time by mitigating the quadratic costs of large self-attention computations.

1 Key Information to include

- TA mentor: Anna Goldie
- External collaborators (if no, indicate "No"): No
- External mentor (if no, indicate "No"): No
- Sharing project (if no, indicate "No"): No

2 Introduction

Over the past few years, innovations in compute and storage, as well as the digitization of information and communication have created a surge in the amount of data we produce and collect. Besides just image and text data, we are also seeing new forms of data emerge, such as genomic data, VR/AR data, and biometric data, to name a few. Because the rapid rate of data production will only increase over time, there is a pressing need to find the most optimal ways of compressing and decompressing data for the most efficient storage and transfer of these data.

As we know from information theory, the ability to predict data and the ability to compress data are closely correlated. Accurately predicting data involves learning patterns and redundancies in the data, and these findings are useful when implementing compression techniques. Thus, advances in natural language process subsequently lead to opportunities for advances in compression. Over recent years, transformer-based models have proven to be extremely effective in various natural language processing tasks such as language translation, semantic parsing and language modeling.

In this project, we specifically investigate whether we can leverage the next-token prediction of an autoregressive transformer model to achieve superior compression ratios for the task of lossless text compression. We also utilize additional data compression algorithms on top of a transformer-based model and compare its performance to our baselines.

3 Related Work

Existing work has been done investigating using RNN-based models for compressing sequential data. For example, in Goyal et. al's paper on DeepZip [1], they demonstrate that using RNNs and arithmetic encoding, DeepZip performed 20% better in compression size compared to Gzip, a standard baseline for text compression, on text and genomic data. However, their approach relied on overfitting the RNN on the input data and therefore required including the model weights as part of the compressed representation. On large files this is not a significant issue because the increase in file size due to the model is small relative to the decrease in file size due to superior compression. However, on small files, including the model weights erases any compression gains achieved by the model.

While DeepZip used an RNN and tokenized at the character level, Bellard's [2] neural text compressor used a transformer model and multi-character tokens. Because subword-level tokens generally lead to better next-token predictions than character-level tokens, it is unsurprising that Bellard's compressor achieved better compression performance. It produced compressed file sizes 50% smaller than Gzip. Similar to DeepZip, the compressed file includes the model weights.

Additionally, previous work on image compression has demonstrated the opportunites to leverage deep-learning approaches to achieve lossy compression. Theis et. al [3] demonstrated the the use of an convolutional auto-encoder as an image compressor. The image would be passed through the encoder to generate a latent representation which constituted the compressed image. Then the latent representation would be passed through the decoder to create obtain an lossy copy of the original image.

While existing research has demonstrated success of using neural networks to compress text data, we find an opportunity to improve upon it. Our implementation does not require transmission of model weights. This means that the minimum compressed file size is much smaller, thus making it possible to achieve good compression results on much smaller files compared to previous implementations.

4 Approach

4.1 Variable-length integer (varints) encoding/decoding overview:

Varints is an algorithm that compresses fixed-length integers into variable-length integers to save space. For example, instead of storing every integer as 4 bytes, with varints, we can store a 100 in one byte, a 300 in 2 bytes, and 1,000,000 in 3 bytes. This means we can save a lot of space if we only handle mostly small numbers and only sporadically very big numbers. This the key idea we leverage in our approach that we discuss later in this section. We invite you to read this resource for more info on varint.

Our implementation of varint encoding and decoding adapted an existing varint implementation by Bright-Tools [4].

4.2 Arithmetic encoding/decoding overview:

Arithmetic encoding is a type of entropy encoding that is used to compress and decompress a stream of data. Frequently used characters are stored with fewer bits and less frequent characters are stored with more bits. In contrast to other forms of entropy encoding, such as Huffman coding, arithmetic encodes input text into a single number rather separating the input into discrete symbols, each represented with a code. The arithmetic encoder takes in two inputs:

1. A probability distribution over the next symbol
2. The next symbol to encode

Adapative arithmetic encoding allows the probability distributions fed into the encoder to change as more text is read in. This still results in lossless compression because the tables will symmetrically change during decoding. We invite you to read Witten's paper on arithmetic encoding for more details [5].

Our implementation of adaptive arithmetic encoding and decoding adapted an existing open-source implementation by Project Nayuki [6].

4.3 Model:

We used a pretrained GPT2 model as our transformer of choice, since its extremely effective ability to generate "next-word" predictions fit perfectly with our use case.

4.4 Compression:

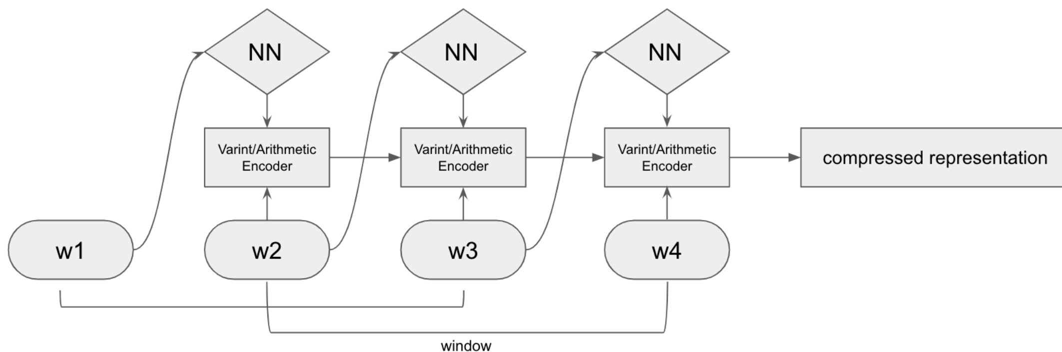
We feed our transfomer-based model a document to compress word-by-word. For example, at word w_i, we pass the model the previous $w_{i-k'}, ...w_{i-2}, w_{i-1}$ words in order and have the model generate a distribution over the vocab for next-word predictions, where k' is the number of words before w_i in the current window. We sort the next-word distribution from most likely to least likely. We then iterate through these next-word candidates until we find w_i. The index of this word in the sorted list is then passed through our variable-length integer encoding algorithm or our arithmetic encoding algorithm, and this encoding is stored as part of the compressed representation.

For variable-length integer encoding, the better our model is able to predict the next word, the smaller the ranking, and the smaller the varint encoding. For arithmetic encoding, the better our model is able to predict the next word, the more frequent small indices will appear, and thus their arithmetic encoding will be small. This means that model performance is directly linked to compression performance.

We also implementing a windowing approach in our compression. For a given document of length l, the document is padded the document to a length $l' = \lceil \frac{l}{k} \rceil \cdot k$, thus guaranteeing that $l' \mod k = 0$. The document is then split into $\frac{l'}{k}$ windows Win_1, Win_2 ... $Win_{\frac{l'}{k}}$. Each window is run through the encoder, where $WinEncoded_i = Encoder(Win_i)$. Then, the encoded representations are concatenated such that $Result = [WinEncoded_1; WinEncoded_2; ...WinEncoded_{\frac{l'}{k}}]$.

Using windowing is necessary for two reasons. First, it prevents computation cost from scaling quadratically with file size. For a file of n tokens, without windowing the time complexity for self-attention is $O(n^2)$. On the other hand, with a window of size k, the transformer only computes attention on a maximum of $k - 1$ tokens at a time, leading to a self-attention time complexity of $O(k \cdot n)$. Windowing is also necessary because text files can be far longer than the 1024 token limit of GPT-2. Windowing allows for a large text file to be split into multiple small windows, thus allowing for compression of arbitrarily-large documents.

Figure 1: Compression Framework



4.5 Reconstruction:

During reconstruction, we simply run the encoder in reverse: we take our compressed representation and pass it through the varint or arithmetic decoder to get the index of the model's sorted next-word prediction. Thus, we can pass each decoded word through the model again to fetch each subsequent word.

4.6 Baseline:

We used two baselines: Gzip and statistical encoding.

1. **Gzip**: Gzip's compression approach is a mixture of Lempel-Ziv encoding and Huffman encoding. Lempel-Ziv encoding algorithms compress data by replacing repeat occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. Huffman encoding is another type of entropy encoding, where more common symbols are generally represented using fewer bits than less common symbols.

2. **Statistical encoding**: For a statistical encoding baseline, we simply took the input text, tokenized the text, and passed the entire array of tokens through an arithmetic encoder as a string. In contrast to how we used arithmetic encoding described in our "Approach" section, this baseline does not depend on a transformer's output in any way. Thus, compression is not influenced by neural network performance.

5 Experiments

5.1 Data

We used the CNN/Dailymail dataset to evaluate our compression. It is an English-language dataset containing just over 300k unique news articles as written by journalists at CNN and the Daily Mail. We selected the first 1000 articles from this dataset for our experiments.

5.2 Evaluation method

We evaluated compression ratio (Uncompressed Size / Compressed Size) and compared between our neural approaches and baselines. We calculated the mean, minimum, maximum and standard deviation for the compression ratio over the 1000 articles and compared between our approaches. We also evaluated the compression speed between our neural approaches and baselines.

5.3 Experimental details

We finetuned a pre-trained GPT2 model on the task of text generation on the wikitext2 dataset. Fine-tuning hyperparameters included: a learning rate of 2e-05, training and evaluation batch size of 8, an Adam optimizer.

We implemented our neural compressor with two different data compression approaches: one with varint encoding and one with arithmetic encoding. We evaluated a compressor for window sizes $32, 64, 128$ with a batch size of 32 and window size 256 with a batch size of 8.

5.4 Results

In terms of compression ratio, we see from Figure 2 and Table 1 that our neural approaches surpass our baselines. Both varint and arithmetic encoding neural approaches led to >4X compression of input text, compared to 2X for our Gzip and statistical encoding baselines.

Figure 2: Compression Ratio over 1000 articles

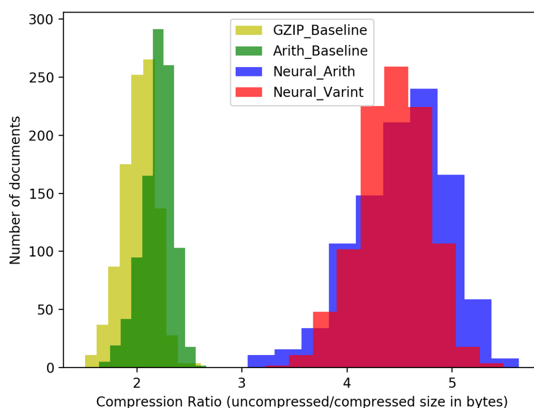


Table 1: Compression Ratio (Uncompressed/Compressed): no windowing, averaged over 1000 articles

	Mean	Min	Max	STD
Ggzip_Baseline	2.02	1.51	2.61	0.16
Arith_Baseline	2.20	1.64	2.66	0.15
Neural_Varint	4.44	3.22	5.49	0.32
Neural_Arith	4.52	3.05	5.63	0.44

While neural approaches lead to superior compression ratios, in terms of compression speed, neural approaches are much slower. With a window size of 64 tokens, our neural compressor is approximately 4,500 times slower than Gzip.

The figures below further illustrate how window size affects neural compression speed. Using smaller windows led to better compression speed with the exception that speed was slightly better for $k = 64$ compared to $k = 32$ (Fig 3). With $k > 64$, compression speed appears to increasing super-linearly. Average time per document for $k = 128$ is approximately 1.5 times slower than $k = 64$, but average time per document for $k = 256$ is approximately 2.5 times faster than $k = 128$.

While changes in window sizes had significant effects on speed, the same cannot be said for compression ratio. Regardless of window size, compression ratio was approximately $4x$, with an improvement from 3.86 to 4.23 between $k = 32$ and $k = 256$ (Fig 4). Figure 5 shows the trade-off of compression ratio vs time.

Figure 3: Window Size vs Average Time Per Document

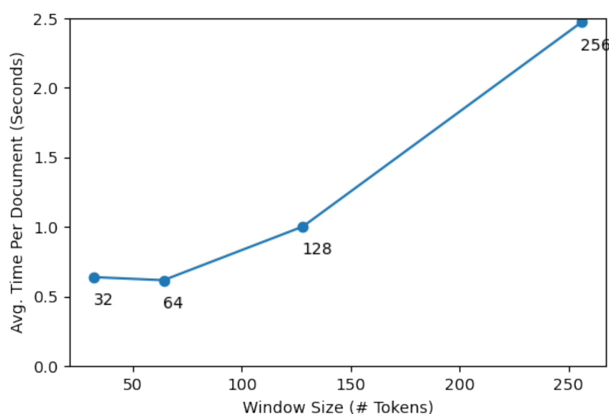


Figure 4: Window Size vs Compression Ratio

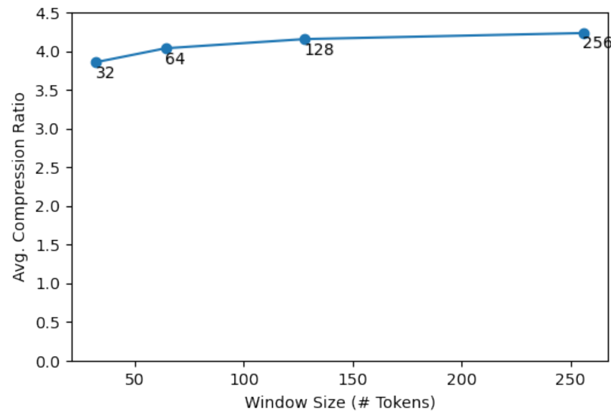
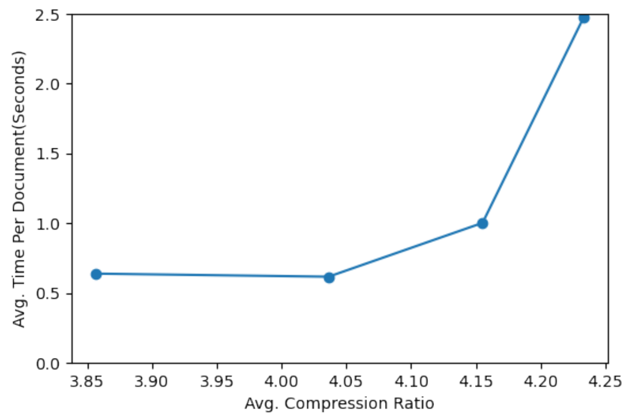


Figure 5: Compression Ratio vs Average Time Per Document



While Gzip only achieves optimal results for large files, original file size does not appear to have a significant impact on the compression performance in our neural approach (Fig 6). Compression ratio only increases by approximately $.5$ between the smallest and largest file for each of $k = 32, 64, 182, 256$. However, Figure 7 does show that the variance of compression ratio decreases as file size increases.

Figure 6: Neural Compression: Original File Size vs Compression Ratio (Linear Fit)

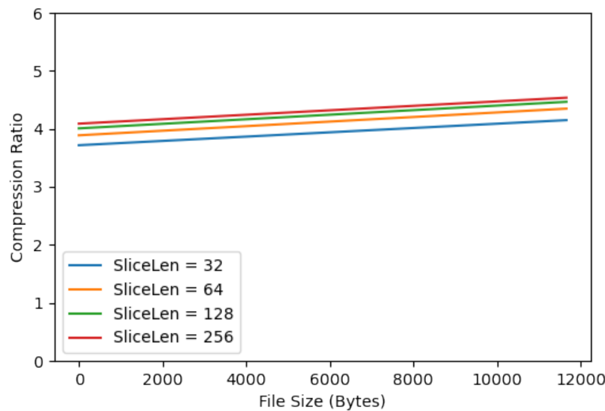
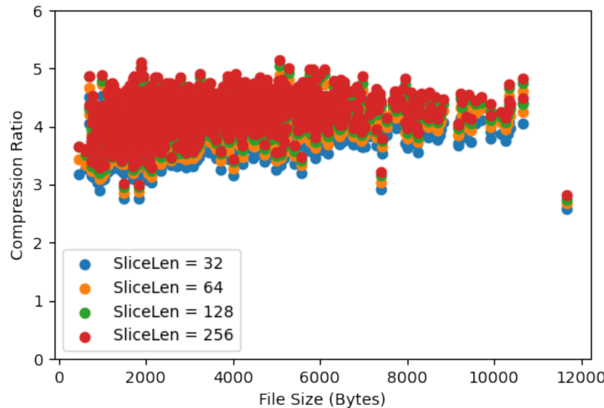


Figure 7: Original File Size vs Compression Ratio (Scatter)



6 Analysis

Our results demonstrate the inherent trade-off between compression ratio and compression speed. While neural compression approaches are superior in compression ratio, they are inferior in compression speed. However, the superior performance of our neural compression approaches could be useful for use cases that do not depend as much on compression speed, for example on smaller documents.

Our experimentation with various window sizes suggests that a window size of $k = 64$ is optimal because it has the fastest compression speed and has a compression ratio only negligibly smaller than that of large window sizes. It is not totally clear why $k = 64$ outperformed $k = 32$ but we postulate that this may be because smaller window sizes lead to less efficient batching: in our implementation, two documents d_1, d_2 are only eligible to be batched together if for the lengths l_1, l_2 it is the case that $\lceil \frac{l_1}{k} \rceil = \lceil \frac{l_2}{k} \rceil$. Thus, reducing the value of k leads to fewer documents being batched together.

We postulate two possibilities for why the variance of compression ratio reduces for larger files. First, it may be due to the law of large numbers: the compression ratio for small sequences may vary from the mean, but as length of the document increases, compression ratio reverts to the mean. Second, it may simply be due to experiment design: our dataset includes fewer large files, so the reduction in variance may simply be due to there being fewer samples.

7 Conclusion

Our transformer-based lossless text compressor results in a 2X better compression compared to Gzip, the industry standard for text compressors, and our statistical encoding baseline. We were able to achieve these results through both our varint and arithmetic encoding/decoding approaches. We also found, however, that it takes 3 orders of magnitude more time to compress using our approach compared to Gzip, even with windowing input text.

Future work could benefit from doing a closer analysis of other bottlenecks affecting computation time, since the largest weakness of our approach is how slow it is. There are two principle steps that could be taken to improve runtime. First, we could change the batching strategy to achieve better parallelism. Second, we could reduce model complexity by reducing the number of layers and replacing dot product self-attention with synthesizer self-attention.

Beyond encoding/decoding the model's next-word index with varint and arithmetic encoding, it could be worthwhile to explore other encoding/decoding approaches as well. We also think exploring other ways to improve transformer's performance on next-word prediction could be useful, since model performance and compression performance are closely linked. For example, we could create a variational autoencoder using a pretrained encoder and decoder and then fine-tune the autoencoder on document reconstruction using categorical cross-entropy loss. The original document could be transformed by the encoder into a latent space representation, which could be subsequently passed

into the decoder. This could improve next-word predictions and consequently improve compression performance.

References

[1] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, “DeepZip: Lossless data compression using recurrent neural networks,” *arXiv:1811.08162 [cs, eess, q-bio]*, Nov. 20, 2018. arXiv: 1811.08162. [Online]. Available: http://arxiv.org/abs/1811.08162 (visited on 02/08/2022).

[2] F. Bellard, “Lossless data compression with neural networks,” p. 11,

[3] D. Liu and G. Liu, “A transformer-based variational autoencoder for sentence generation,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, ISSN: 2161-4407, Jul. 2019, pp. 1–7. DOI: 10.1109/IJCNN.2019.8852155.

[4] bright-tools, *Variable length integer encoding*, original-date: 2017-02-11T20:20:10Z, Jul. 22, 2021. [Online]. Available: https://github.com/bright-tools/varints (visited on 03/14/2022).

[5] I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmetic coding for data compression,” *Communications of the ACM*, vol. 30, no. 6, p. 21, 1987.

[6] Nayuki. “Reference arithmetic coding,” Project Nayuki. (Jun. 28, 2018), [Online]. Available: https://www.nayuki.io/page/reference-arithmetic-coding (visited on 03/14/2022).