*MAJOR TECHNICAL PROJECT ON*

# DEEP LEARNING FOR TEXT COMPRESSION

*MTP REPORT*

*submitted by*

**ANVAY SHAH**
**B17078**

*for the award of the degree*

*of*

**BACHELOR OF TECHNOLOGY IN**
**COMPUTER SCIENCE AND ENGINEERING**

**SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING**

**INDIAN INSTITUTE OF TECHNOLOGY MANDI, MANDI**

**June 2021**

# Abstract

Text Compression has a number of applications today in the reduction and storage of data. Text compression must be lossless as we require an exact reconstruction of data. One of the most popular text compression algorithms today is Gzip, whose algorithm consists of a sliding window dictionary technique coupled with Huffman coding. Gzip tries to find a balance between compression time and compression size, and takes quite a short amount of time. In the context of Data storage however, even a slight reduction in data size is very useful at the expense of a large amount of time. For this purpose people are investigating the use of deep learning for text compression. Deep learning based methods are significantly slower than Gzip but show potential for a much better compression performance.

We have identified some promising work in this area, namely DeepZip: Lossless Data Compression using Recurrent Neural Networks, which uses Recurrent Neural Networks like LSTMs and GRUs coupled with Arithmetic coding to compress text. We have built on their work, using Transformer models instead of Recurrent Neural Networks. Our experimental results on datasets show that these models outperform Gzip and show significant potential for further improvement.

**Keywords:** *Compression, Lossless, Text, Deep Leaning, Transformers.*

# Contents

# Chapter 1

# Introduction

Data compression is the process of creating an encoding or a compressed representation that uses a lesser number of bits than the original data in a either a lossless or lossy manner. Lossless data compression is a perfectly reversible process that finds redundancies in the data and preserves the information content while reducing the redundancies and thereby the number of bits. Lossy compression is used for images, where we do not require an exact reconstruction of the image. Text, however can be less forgiving. Our problem of focus here is lossless text compression. Text can be compressed losslessly by assigning codewords to the sequences of characters based on their frequency – sequences of characters which are more frequent are assigned shorter codewords and sequences of characters which are less frequent are assigned longer codewords.

The theoretical limit for compression of a sequence is given by its entropy. The entropy is the expected value of the information of a random variable. As we must preserve information in lossless compression, it is intuitive that entropy gives the maximal compression limit. Huffman coding [1] and Arithmetic coding [2] are popular techniques that create optimal prefix codes, i.e. they achieve this theoretical optimal limit. But their implementation requires knowledge of probability distribution of the sequence, which is not always available and rather difficult to estimate.

Popular compression algorithms like Gzip (`https://www.gzip.org/`) use other techniques like Lempel Ziv coding (LZ77 and LZ78) [3] in combination with Huffman coding [1]. LZ77 is a dictionary coding technique - it finds repeated occurrences of subsequences in the string to be compressed and replaces them with a pointer to the first occurrence. It uses a sliding window technique to locate these repeated occurrences. Gzip then uses Huffman coding on the output encoding produced by LZ77 to assign shorter codewords to more frequent encodings. There is a time - performance trade-off to the Gzip algorithm - it requires more time to perform better compression. Along with Gzip, several other techniques like NNCP [4], cmix [5], 7zip (`https://www.7-zip.org/`) are also widely used today.

As mentioned above, estimating the probability distribution of input data is a challenging task. If we could perfectly estimate the probability distribution we could use optimal codes to achieve the entropy limit for data compression. However, if we estimate distribution P with distribution Q, we incur a penalty that is equal to the relative entropy of P and Q on top of the entropy. Several deep learning frameworks like Variational Autoencoders (VAEs) [6] and Generative Adversarial Networks (GANs) [7] attempt to do this very task of estimating distributions in order to generate new data that belongs to the same class of distribution. GANs have achieved impressive performance on tasks such as generating faces indistinguishable from real images.

Recurrent Neural Networks (RNNs) are adept at learning patterns in long sequential data. They are used in tasks like language modeling and machine translation. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks are better than vanilla RNNs at caputuring long term dependencies in sequences as they solve some of the problems faced by vanilla RNNs, such as vanishing gradients. They process words in sentences sequentially and retain past information through hidden states. The state-of-the-art deep learning model in sequence processing, however, is the Transformer [8]. A Transformer processes sentences non-sequentially as a whole and uses the mechanism of Self Attention [8] on an input with a positional embedding to outperform LSTMs and GRUs on the above mentioned tasks.

In light of their recent success [9] in tasks of sequence processing, we have used Transformers to estimate the probability distribution of input text. We have then used Arithmetic coding to compress the text based on the predicted distribution. We have experimented with different model architectures and tabulated results for varying file sizes from different datasets, including the enwik9 dataset [10], the IMDB reviews dataset [11] and a number of books and novels. We have consistently achieved a better performance than Gzip on these datasets.

# Chapter 2

# Background and Related Work

The Large Text Compression Benchmark [12] is a competition that ranks lossless text compression algorithms based on their performance on the enwik9 dataset [10]. This is a dataset of size $10^9$ bytes which consists of an XML dump of the English version of Wikipedia. The best performing algorithm that compresses the enwik9 dataset under certain hardware and time constraints receives a prize of 500,000 euros, known as the Hutter Prize [13].

We have identified a promising algorithm in this area, namely DeepZip: Lossless Data Compression using Recurrent Neural Networks [14]. The authors of DeepZip [14] have used bi-directional Long Short-Term Memory (biLSTM) and bi-directional Gated Recurrent Unit (biGRU) networks for the purpose of probability distribution estimation and then used Arithmetic coding for the compression task. DeepZip [14] has achieved a better performance than Gzip on specific datasets like Human chr1 dataset, C. Elegans chr1, C. Elegans whole genome and a few others. These datasets contain DNA sequences or genomics data which has a limited alphabet size (for example, DNA sequences have an alphabet of 5 characters; A,T,G, C and N only) and is easier to compress. We have tested the Deepzip models, BiGRU and BiLSTM on more generalized datasets and built on their work, experimenting with various Transformer models.

Earlier this year (in early 2021) Fabrice Bellard published the second version of NNCP [4], which is the current best performing algorithm in the Hutter Prize Challenge, with a compressed size of 110.2 MB on the enwik9 dataset. The approach used in this paper has many similar elements to ours, notably the use of Transformers and Arithmetic encoding.

# Chapter 3

# Work Done

## 3.1 Literature Review

I studied several topics in information theory such as types of codes and the bounds on optimal code length from Elements of Information Theory [15] as a background to better understand the field of Data Compression. I studied the working of optimal or close to optimal length coding techniques like Huffman coding, Arithmetic coding, Shannon coding, Fano coding and the proofs of their optimality. I read about Universal source coding, which explores creating encodings without knowledge of the distribution of the source. Universal source coding techniques can be used to compress data from any source distribution. For example, Lempel Ziv coding [3] is used to encode data that is difficult to model, such as text. The penalty paid in using Universal source coding techniques over techniques such as Huffman coding where the source distribution is known, is an increase in block length and a more complex encoder / decoder model.

I read research papers and research articles about previous work in this area, namely DeepZip [14] and BitSwap: A Deep Learning Approach to Data Compression [16].

## 3.2 Implementation Details and Hardware Used

I implemented the Transformer models in python on Google colab, using Tensorflow version 2.5.0 and a Tensor Processing Unit (TPU) as the hardware accelerator.

The DeepZip code was written in Tensorflow version 1 and required older versions of software. At first, I attempted to run the code on my laptop but I faced a myriad of issues in configuring the required software packages and graphics card due to incompatible versions. I was able to run it on a friend's computer using docker. As this was only a temporary solution, I gained access to the IIT Mandi High Performance Cluster (HPC).

The HPC does not have docker and requires usage of Singularity containers instead. However, the version of CUDA required by the code was different than the version on the HPC and Singularity could not bridge the gap. Additionally, I was struggling to implement a Transformer model in Tensorflow version 1 as all the required Keras layers for Transformers are written in Tensorflow verion 2. I decided that the only way to go was to port the code into Tensorflow version 2. But I discovered that the HPC could only support Tensorflow version 2.0.0 and I required at least version 2.4.0 to access the required Keras layers. I then ported the code to Google Colab.

Google Colab solved all the problems, and more. The Google Colab GPU was slightly faster than the HPC. Google Colab also had an option to use their new state-of-the-art TPUs instead. Once configured correctly, they are blazing fast for training neural networks. I was able to optimize my code for the TPU and this was a game changer. The HPC could run an epoch for the biGRU network in about 20 minutes. The TPU changed this time to a mere 25 seconds. It is faster than the HPC by a factor of almost 50. In all, this entire struggle was a huge learning experience. In the following sections, I have provided the algorithmic details of the code.

## 3.3 Datasets Used

### 3.3.1 Enwik9 Dataset

As mentioned in Chapter 2, this consists of data from the English version of Wikipedia. It has a total size of $10^9$ bytes. I split the dataset into 5, 10, 15 and 20 MB chunks. I used several files from each size category to run the tests. This dataset has an average alphabet size of close to 200.

### 3.3.2 IMDB Reviews Dataset

This conists of film reviews, categorized as positive and negative reviews. I concatenated all of these and stripped the files of all special characters. I switched all uppercase characters to lowercase, thus reducing the alphabet size to 40. I took out two chunks of this dataset for testing purposes, one of size 30 MB and the other of size 40 MB.

### 3.3.3 Books

In addition to the above large datasets, I used my models to compress several books such as Harry Potter 1 and 2, Dracula, War and Peace and so on. These had a size of between 0.5 to 3 MB.

## 3.4 Transformer Models

I built the Transformer model described in Figure 3.1 for my experiments. The model consists of a Token and Postional Embeddings layer, followed by N Transformer blocks (structure given below). The number of Transformer blocks N were varied. Three different values were experimented, N = 1, N = 3 and N = 6. Next is a pooling layer, followed by a dropout layer, followed by a dense layer.

The Token and Postional Embeddings layer is crucial in a Transformer. The purpose of the positional embedding is so the model has information regarding the sequence of the input whereas the purpose of the token embedding is to convert each word into a dense vector of fixed size. The input dimension for the positional embedding is equal to the input sequence length K and the input dimension for the token embedding is equal to the alphabet size of the input. Both of the embeddings have an output dimension equal to a hyperparameter H, which was set as 32. The output of the Token and Postional Embeddings layer is the sum of these two embeddings.

I built a Transformer block with the structure given in Figure 3.2. The most important layer in a Transformer is the Multi Head Attention layer. This layer works on the mechanism of Self Attention [8]. I have used two attention heads and the input and output dimensions

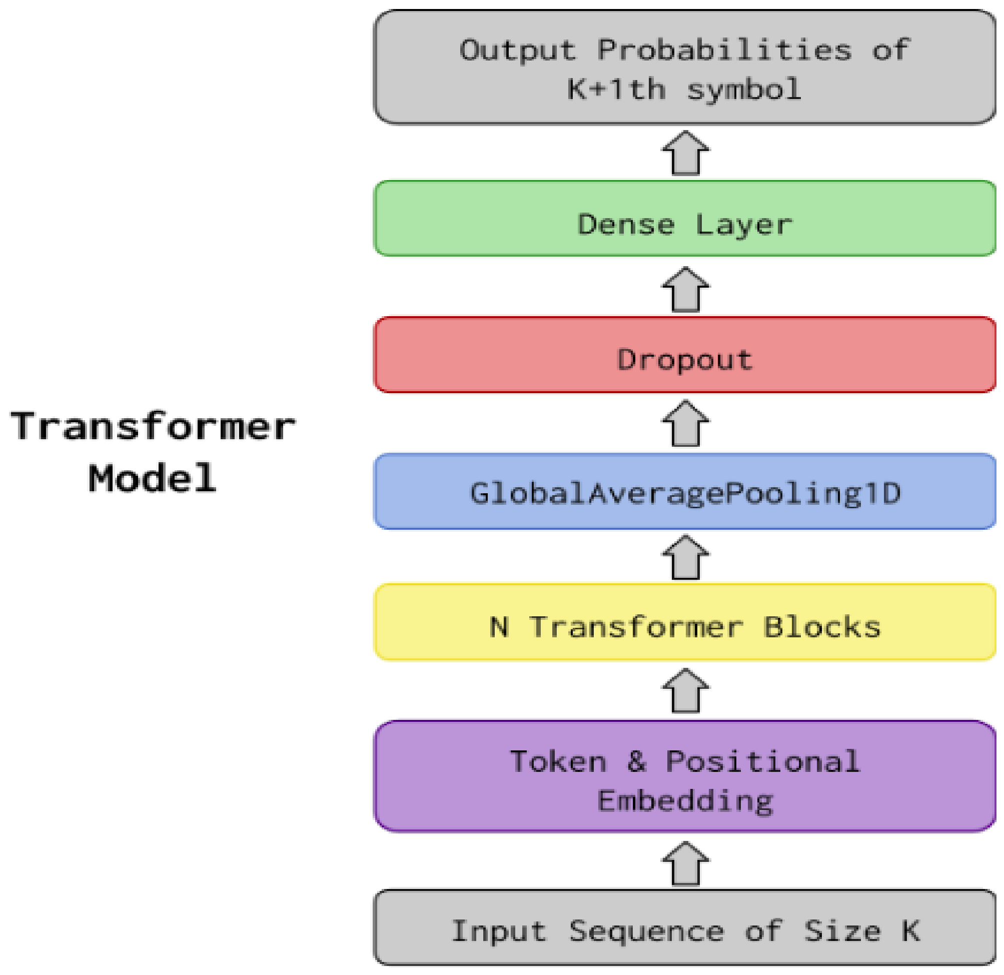


**Fig.** 3.1: Structure of a Transformer Model

are both equal to H. Following this we have a dropout layer. The outputs of the dropout layer are summed with the inputs and normalized. Next is a Feed Forward Network with two dense layers. Both have H hidden units and use a ReLU activation [17]. Next is a dropout layer, whose outputs are summed with the inputs of the Feed Forward layer and normalized again.

The final layer of the Transformer model is a dense layer with a softmax activation. The output dimension is equal to the alphabet size of the input sequence and the input dimension is H. The outputs are probabilities of the K+1$^{\text{th}}$ symbol in the sequence.

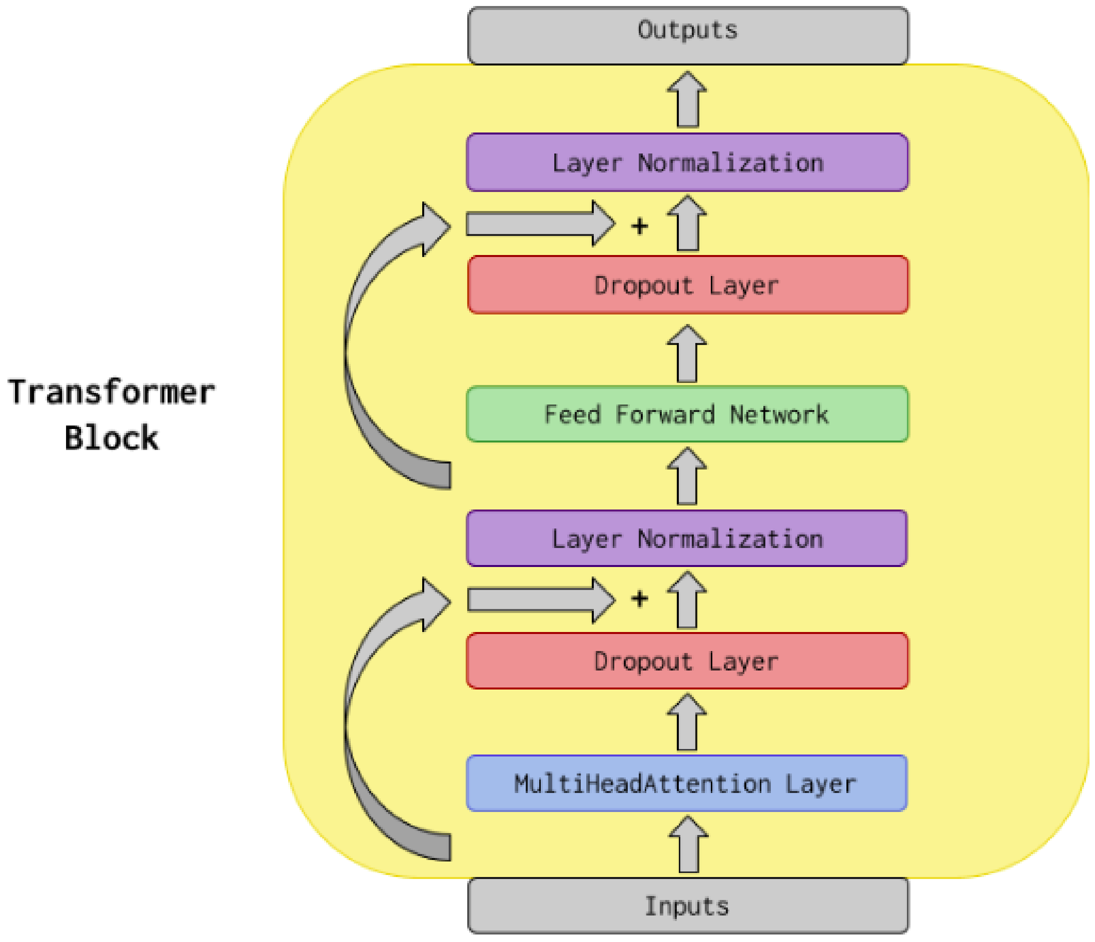


**Fig.** 3.2: Structure of a Transformer Block

## 3.5 Training

As the only input to a trained model is the training data, we attempt to overfit the model on the input sequence. We sample subsequences of length K+1 from the sequence to be compressed. The first K symbols are the input and the corresponding label is the $(K+1)^{\text{th}}$ symbol. To train the model, the optimizer used was Adam [18] and the loss function was categorical cross-entropy. The hyperparameter K was chosen to be 64. The data was trained over 20 epochs with a batch size of 1024 and the batches were reshuffled on each epoch to aid convergence.

For files larger than 10 MB, the Google Colab TPU buffer was running out of memory. To compress larger files, I had to split the data into 2-4 large batches and split those into smaller minibatches of size 1024. Each large batch was trained for a single epoch in succesion, with shuffling.

## 3.6 Compressor

The compressor consists of a probability predictor model which uses Arithmetic coding to encode each symbol. The encoding is based on a conditional probability estimate of that symbol based on the previous K symbols. This estimate is provided by the trained Transformer.

## 3.7 Decompressor

The decompressor performs symmetric operations to the compressor. It uses the same probability predictor model to output the same probability estimates based on the previous K decoded symbols. This estimate is used while decoding the encoded string using Arithmetic decoding. For both the compressor and the decompressor, the initial K symbols are encoded by using a uniform distribution as the prior.

# Chapter 4

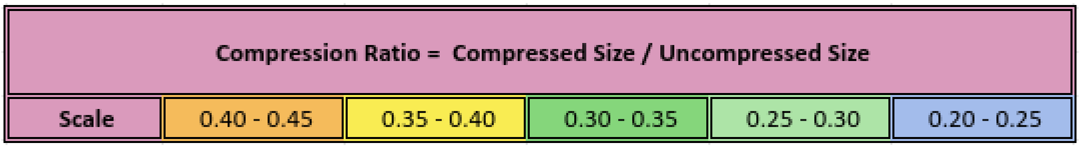
# Experimental Studies and Results

## 4.1 Compression Results on enwik9 Dataset

I broke down the enwik9 dataset [10] into chunks of sizes 5 MB, 10 MB, 15 MB and 20 MB. I compressed these files using different models - 6 Layer Transformer, 3 Layer Transformer, 1 Layer Transformer, biGRU. I also compressed the files with Gzip and 7zip. The results are compiled in Figure 4.2. The colour scheme for Figure 4.2 is shown in Figure 4.1. The leftmost column, titled 'enwik9 Dataset' shows which chunk of the dataset the file is taken from. For example, 0-10 MB refers to the first 10 MB of the dataset and 10-20 MB refers to the next 10 MB. Note that the dimension of the layers H in the Transformers is 32.

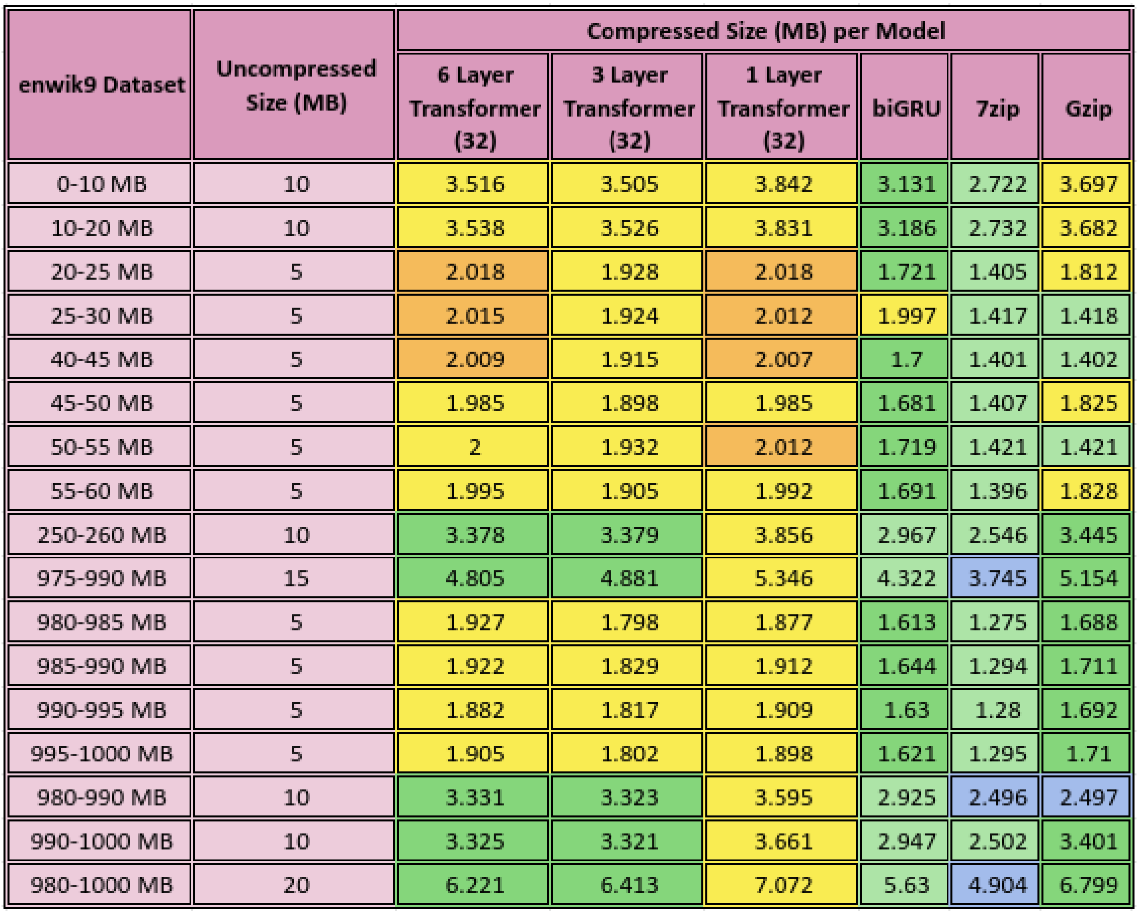
We observe that for 5 MB files, the 3 Layer and 1 Layer Transformers perform better than the 6 Layer Transformer. This is because the compressed size of a file includes the size of the Transformer model used. The size of the Transformer models are approximately 134 Kb for the 1 Layer model, approximately 252 Kb for the 3 Layer model and approximately 423 Kb for the 6 Layer model. The 5 MB files are small enough that the model size is enough to make a significant contribution to the compressed size. Thus, for 5 MB files, the 6 Layer Transformer performs the worst out of the three even though it has the best compressive performance, as it has a large model size.

For 10 MB files and above, the 6 Layer Transformer outperforms the other Transformers. We also observe that it outperforms Gzip. We notice that the biGRU model outperforms all all the other algorithms, except for 7zip, which consistently achieves the best compression performance out of the lot. Thus we conclude that for file sizes up to 20 MB, the 6 Layer Transformer and biGRU outperform Gzip but not 7zip.



| Compression Ratio = Compressed Size / Uncompressed Size | | | | | |
|---|---|---|---|---|---|
| Scale | 0.40 - 0.45 | 0.35 - 0.40 | 0.30 - 0.35 | 0.25 - 0.30 | 0.20 - 0.25 |

**Fig.** 4.1: Colour Scheme for Tables

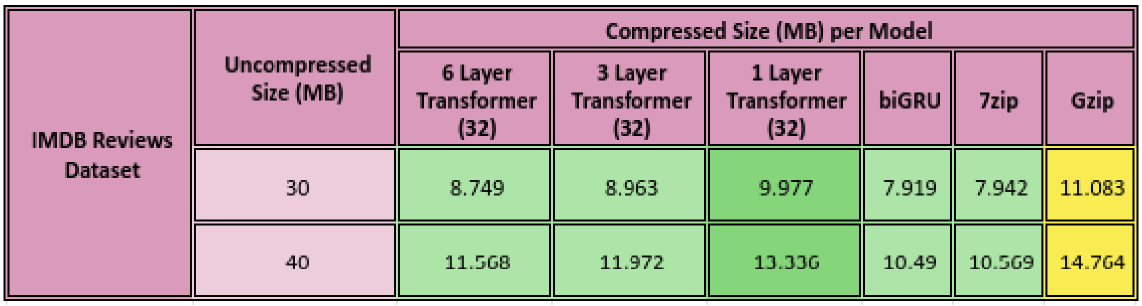


| enwik9 Dataset | Uncompressed Size (MB) | Compressed Size (MB) per Model | | | | | |
|---|---|---|---|---|---|---|---|
| | | 6 Layer Transformer (32) | 3 Layer Transformer (32) | 1 Layer Transformer (32) | biGRU | 7zip | Gzip |
| 0-10 MB | 10 | 3.516 | 3.505 | 3.842 | 3.131 | 2.722 | 3.697 |
| 10-20 MB | 10 | 3.538 | 3.526 | 3.831 | 3.186 | 2.732 | 3.682 |
| 20-25 MB | 5 | 2.018 | 1.928 | 2.018 | 1.721 | 1.405 | 1.812 |
| 25-30 MB | 5 | 2.015 | 1.924 | 2.012 | 1.997 | 1.417 | 1.418 |
| 40-45 MB | 5 | 2.009 | 1.915 | 2.007 | 1.7 | 1.401 | 1.402 |
| 45-50 MB | 5 | 1.985 | 1.898 | 1.985 | 1.681 | 1.407 | 1.825 |
| 50-55 MB | 5 | 2 | 1.932 | 2.012 | 1.719 | 1.421 | 1.421 |
| 55-60 MB | 5 | 1.995 | 1.905 | 1.992 | 1.691 | 1.396 | 1.828 |
| 250-260 MB | 10 | 3.378 | 3.379 | 3.856 | 2.967 | 2.546 | 3.445 |
| 975-990 MB | 15 | 4.805 | 4.881 | 5.346 | 4.322 | 3.745 | 5.154 |
| 980-985 MB | 5 | 1.927 | 1.798 | 1.877 | 1.613 | 1.275 | 1.688 |
| 985-990 MB | 5 | 1.922 | 1.829 | 1.912 | 1.644 | 1.294 | 1.711 |
| 990-995 MB | 5 | 1.882 | 1.817 | 1.909 | 1.63 | 1.28 | 1.692 |
| 995-1000 MB | 5 | 1.905 | 1.802 | 1.898 | 1.621 | 1.295 | 1.71 |
| 980-990 MB | 10 | 3.331 | 3.323 | 3.595 | 2.925 | 2.496 | 2.497 |
| 990-1000 MB | 10 | 3.325 | 3.321 | 3.661 | 2.947 | 2.502 | 3.401 |
| 980-1000 MB | 20 | 6.221 | 6.413 | 7.072 | 5.63 | 4.904 | 6.799 |

**Fig.** 4.2: Compression Results on enwik9 dataset

## 4.2 Compression Results on IMDB Reviews Dataset

I used all the models mentioned in the previous section to compress two chunks of the IMDB reviews dataset [11] of sizes 30 MB and 40 MB. The results are shown in Figure 4.3. Here we observe that biGRU outperforms both 7zip and Gzip for both the files. The 6 Layer Transformer also has a much better compression ratio and outperforms Gzip and comes close to 7zip. We conclude that as file size increases, the deep learning based models improve their performance. We reason that this is because the models can learn more patterns in a larger file.



| IMDB Reviews Dataset | Uncompressed Size (MB) | Compressed Size (MB) per Model | | | | | |
|---|---|---|---|---|---|---|---|
| | | 6 Layer Transformer (32) | 3 Layer Transformer (32) | 1 Layer Transformer (32) | biGRU | 7zip | Gzip |
| | 30 | 8.749 | 8.963 | 9.977 | 7.919 | 7.942 | 11.083 |
| | 40 | 11.568 | 11.972 | 13.336 | 10.49 | 10.569 | 14.764 |

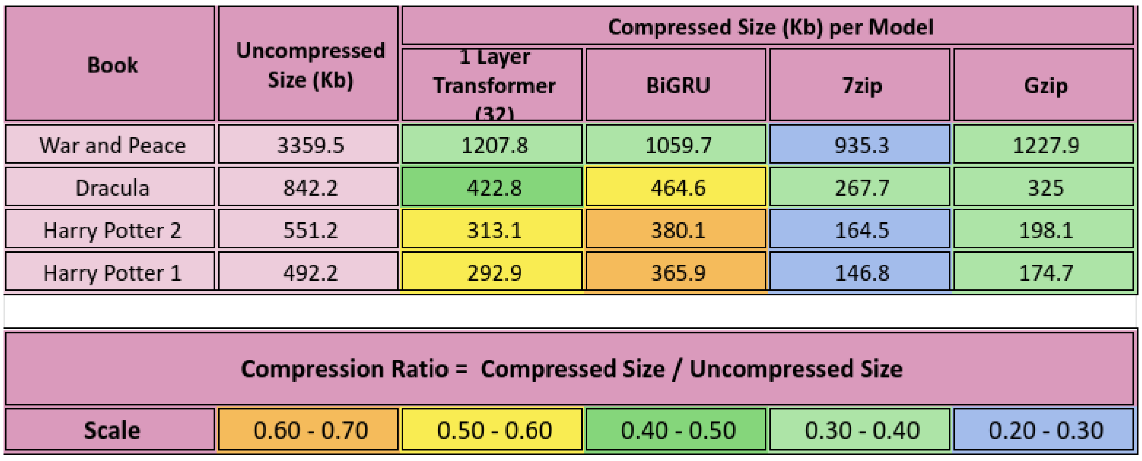
**Fig.** 4.3: Compression Results on IMDB dataset

## 4.3 Compression Results on Books

I used a 1 Layer Transformer, biGRU, 7zip and Gzip to compress a few books. The results are shown in Figure 4.4. These books have a size of around 0.5 MB to 3.3 MB. We conclude that these files are too small for the deep learning based models to show any significant performance.

## 4.4 How Compression Ratio varies with File Size

The results of the above mentioned models over the three datasets are compiled into the graph in Figure 4.5. I was not able to compress files larger than 40 MB due to hardware constraints. Judging by the nature of this graph, I expect that for files that are 100s of MBs in size, the Transformer models may outperform 7zip.



| Book | Uncompressed Size (Kb) | Compressed Size (Kb) per Model | | | |
|---|---|---|---|---|---|
| | | 1 Layer Transformer (32) | BiGRU | 7zip | Gzip |
| War and Peace | 3359.5 | 1207.8 | 1059.7 | 935.3 | 1227.9 |
| Dracula | 842.2 | 422.8 | 464.6 | 267.7 | 325 |
| Harry Potter 2 | 551.2 | 313.1 | 380.1 | 164.5 | 198.1 |
| Harry Potter 1 | 492.2 | 292.9 | 365.9 | 146.8 | 174.7 |

| Compression Ratio = Compressed Size / Uncompressed Size | | | | | |
|---|---|---|---|---|---|
| Scale | 0.60 - 0.70 | 0.50 - 0.60 | 0.40 - 0.50 | 0.30 - 0.40 | 0.20 - 0.30 |

**Fig.** 4.4: Compression Results on Books

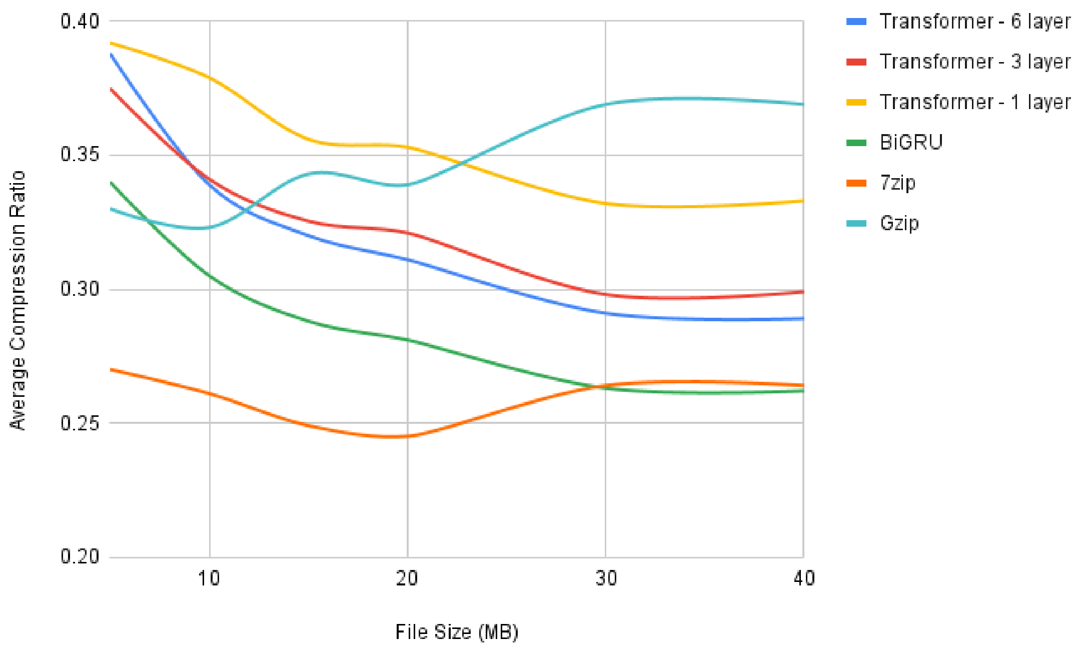


**Fig.** 4.5: Average Compression Ratio (over all Datasets) for Different Models vs File Size

## 4.5 How Model Size varies with File Size

Figure 4.6 shows how a 3 Layer Transformer model size varies with the size of the file being compressed. We observe that it increases slightly till 15 MB and then plateaus. This because the amount of information learnt increases with file size.

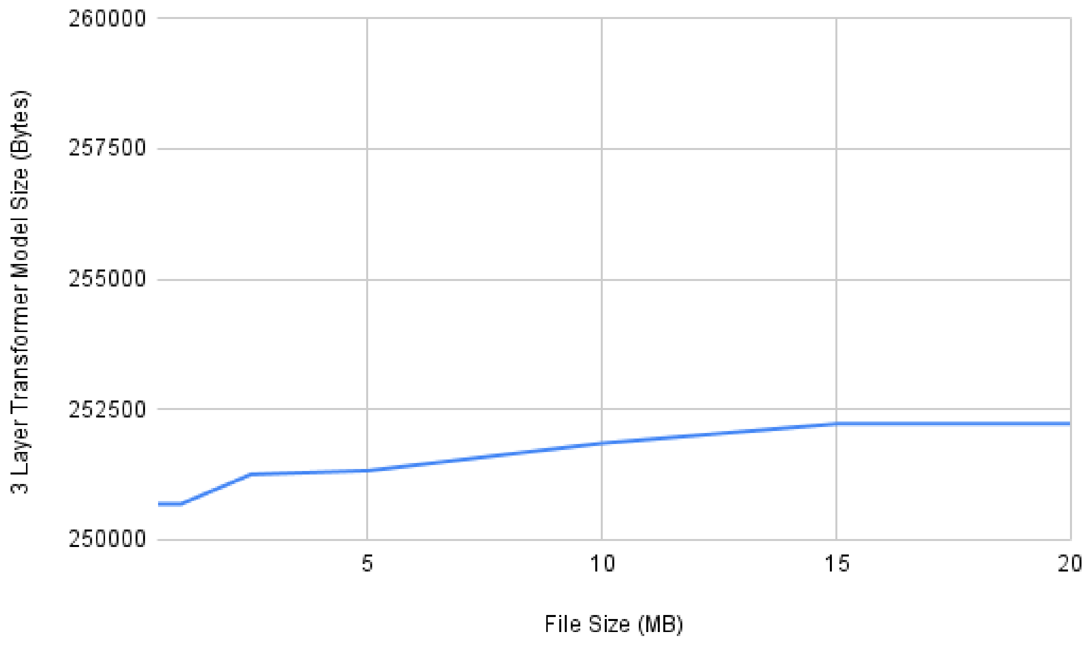


**Fig.** 4.6: Model Size for 3 Layer Transformer vs File Size

## 4.6 How Compression Ratio varies across Transformers

Figure 4.7 shows how the ability of a Transformer to compress increases with number of layers. This leads us to predict that stacking more layers will lead to better compressive ability. This is impractical for small files due to the subsequent increase in model size. Further work and experiments are required to confirm whether there exists a threshold file size above which a 8 Layer Transformer or a 12 Layer Transformer would perform better than the 6 Layer Transformer. More analysis on this subject has been done in the next section.

## 4.7 How Model Parameters of Transformers vary

In all of the above experiments, the hyperparameter H for the dimension of the layers of the Transformer was chosen to be 32. Experiments with this hyperparameter show that the compressive ability of the Transformer greatly increases when H is increased. This is

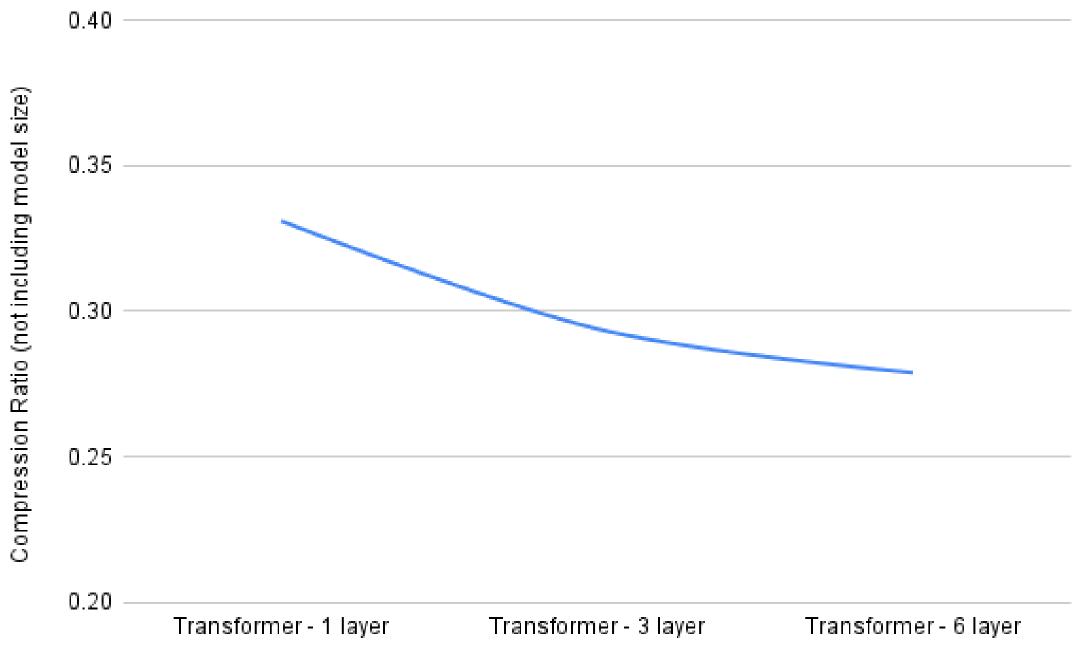


**Fig.** 4.7: Compression Ratio (not including Model Size) for Different Transformer Models

supported by the fact that the Transformer model BERT [19] which achieved state-of-the-art performance on a number of natural language understanding tasks uses value of H = 768. Unfortunately, this increases the number of parameters in the Transformer greatly and subsequently the model size increases greatly as well.

Analysis on the number of parameters used in each layer of the Transformer reveals that the model size increases linearly with an increase in number of transformer blocks and quadratically with an increase in H. The quadratic increase is attributed to the dense layers in the Transformer. The model size becomes of the order MBs when H is larger than 256. Further work and research is required to ascertain whether there exists a threshold file size for which a larger H and a larger number of blocks would outperform the models and compression methods tested in the sections above.

# Chapter 5

# Conclusion and Future Work

The Transformer model built as part of this project showed that it performs quite well, with the performance improving with file size. It outperformed Gzip for files larger than 10 MB and came close to 7zip for files larger than 40 MB. The biGRU model used in DeepZip demonstrated its ability to outperform both. The recent work of Fabrice Bellard in NNCP [4], done in parallel with our work, also shows that Transformers are very powerful in text compression. Thus, we conclude that use of Transformers as a deep learning technique for text compression shows a lot of promise and can outperform conventional text compression algorithms like Gzip and 7zip.

We believe that there is still significant scope of improvement in this area. Performance could be improved by using dictionary coding techniques coupled with a Transformer. Future work is required to analyse whether Transformers with more layers and a larger layer dimension can perform better above a certain large threshold file size.

# References

[1] D. Huffman, “A method for the construction of minimum-redundancy codes,” *Resonance*, vol. 11, pp. 91–99, 1952.

[2] I. Witten, R. Neal, and J. Cleary, “Arithmetic coding for data compression,” *Commun. ACM*, vol. 30, pp. 520–540, 1987.

[3] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theory*, vol. 23, pp. 337–343, 1977.

[4] F. Bellard, “Lossless data compression with transformer,” https://bellard.org/nncp/nncp_v2.1.pdf.

[5] B. Knoll, “Cmix,” http://www.byronknoll.com/cmix.html.

[6] D. P. Kingma and M. Welling, “An Introduction to Variational Autoencoders,” *arXiv e-prints*, p. arXiv:1906.02691, Jun. 2019.

[7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Networks,” *arXiv e-prints*, p. arXiv:1406.2661, Jun. 2014.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[9] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, “Huggingface’s transformers: State-of-the-art natural language processing,” 2020.

[10] M. Mahoney, “About the enwik9 dataset,” http://mattmahoney.net/dc/textdata.

[11] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[12] M. Mahoney, "Large text compression benchmark," http://mattmahoney.net/dc/text.html#1102.

[13] "Hutter prize," http://prize.hutter1.net/.

[14] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, "DeepZip: Lossless Data Compression using Recurrent Neural Networks," *arXiv e-prints*, p. arXiv:1811.08162, nov 2018.

[15] T. M. Cover and J. A. Thomas, *Elements of Information Theory 2nd Edition (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, July 2006.

[16] F. H. Kingma, P. Abbeel, and J. Ho, "Bit-Swap: Recursive Bits-Back Coding for Lossless Compression with Hierarchical Latent Variables," *arXiv e-prints*, p. arXiv:1905.06845, May 2019.

[17] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2018, cite arxiv:1803.08375Comment: 7 pages, 11 figures, 9 tables. [Online]. Available: http://arxiv.org/abs/1803.08375

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.