Neural Network-Based Character-to-Symbol Sequence Translation for Text Compression

Simone Teglia
1836794
teglia.1836794@studenti.uniroma1.it

Tommaso Bersani
1840825
bersani.1840825@studenti.uniroma1.it

Abstract

This paper explores the possibility of the application of neural network models to a sequence to sequence translation task with the aim of text compression. We investigate the feasibility of using deep learning techniques to emulate traditional zipping software, focusing on the Deflate algorithm for its balance between simplicity and efficiency. Our approach involves converting text and zipped files into hexadecimal format to facilitate model training and prediction. We experimented with a range of neural network architectures, including Fully Connected Multi-Layer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and state-of-the-art Pretrained Language Models (PLMs) like BART and T5. We assessed and compared the models' performance leveraging the Levenshtein edit distance metric. Additionally, we discuss computational considerations and outline several non-working approaches. **We release our code in a modular notebook where it's possible to run tests with different models just by setting global variables in the first cell.**

1 Introduction

The goal of this project was to create a model that was able to shrink text turning it into a format compatible with an unzip software, possibly making file compression more efficient. The zipping a text file is a deterministic process, that is supposed to be lossless for formats like ZIP, RAR, 7z, etc., meaning the decompressed file should be identical to the original pre-compressed file, ensuring 100% accuracy in terms of data integrity. Furthermore through the years zipping software have become extremely efficient. Creating a neural network to emulate these softwares presents several inherent difficulties. Neural networks excel at pattern recognition and learning from examples, but struggle with the deterministic, error-free compression and decompression required, as they inherently introduce approximation errors.

1.1 The encoding problem

The first problem that we faced was deciding what type of data feed the neural network with and similarly what type of data getting as output. Given that a zipped file is an unreadable set of bytes that if decoded would result in non-ASCII characters, we decided to convert it into something readable. Therefore converting both the input and the output files into strings was a very intuitive way to convert the data into a format readable and produceable by a model.

The files that we had were text files containing english sentences as input and files created by zipping software as output. We faced the problem of how to encode them in the best possible way to feed them to the neural network. We experimented with several setups, both as input and output. Binary, hexadecimal, base64 were the options that we came up with as formats for the output, whereas plain ASCII and an hexadecimal conversion were the ones for the input.

In the end we decided to give in input the hexadecimal conversion of the original string and as output the hexadecimal conversion of the zipped file. This choice has been made mainly to have a relatively short encoding of both the files without sacrificing too much the understanding of the strings of the model. Indeed, our intuition has been that giving the model inputs and outputs of the same type would have helped the model capturing relationships between them.

1.2 Zipping algorithms

It's interesting to note that "zipping" a file has not an univoque meaning. Indeed there exist several zipping algorithms, some of which (mplode, Deflate64, bzip2, LZMA ...) are implemented in the most diffused zipping softwares. We decided to stick with the Deflate algorithm, which compresses text by combining two key techniques: LZ77 and Huffman coding. First, LZ77 reduces redundancy by replacing repeated strings with pointers to earlier occurrences. Then, Huffman coding further compresses the data by encoding frequently used characters with shorter bit sequences and less frequent characters with longer sequences. This dual approach effectively reduces the size of text files by eliminating repeated patterns and optimizing the representation of each character based on its frequency of occurrence. Deflate is not the most optimized algorithm among the possible ones in terms of shrinking capabilities, nonetheless its logic is fairly simple, thus preserving more links between the file in input and the file in output. This characteristic is obviously convenient in our case as it simplifies the model's work.

1.3 Dataset

Now it's possible to explain the structure of the dataset we created. The base dataset was an IMDB reviews dataset that can be found on Kaggle at this address. We created a new dataset with two columns:

Table 1: Dataset

text_hex	**deflate_hex**
hex encodings of the reviews truncated to a maximum length	the hex encodings of the zipped txt files containing the reviews truncated to a maximum length

In order to test the models we created or finetuned with several setups, we decided to create several versions of the dataset, imposing the strings that had to be zipped to be of a fixed number of words. The version we sticked to in the end were two versions, where the reviews were truncated respectively to the fourth and to the eighth word. One last dataset has been created, which is a variant of the shorthex2hex. We decided to create it because we noticed a pretty important similarity between the first words of the dataset. This is because we have used a dataset of film reviews and we've seen that there are many reviews that starts in the same way ("I saw this film...", "This film is..."). This made the zipped strings also be very similar and presumably easier to predict for the model. Thus this last one notebook has still a fixed number of words, but the words are taken in random positions from the original string.

2 Baseline

The first step in the implementation of our project is the creation of a non trivial baseline. The idea behind the baseline has been to approach the task as a sequence multiclass classification task, in which each character could assume 17 different classes (the 16 hex characters plus a padding special character that we indicated as "P").

Therefore we created a Fully Connected MLP model that takes in input a whole sequence and outputs a whole sequence in a single shot. In order to do this, each sequence - both the input and the output ones - has been padded in order to reach a fixed length (that depends on the dataset we are using). Once padded, each sequence is one hot encoded character by character into tensors in order to be feedable to the network.

Let's briefly recap the structure of the model. It takes in input tensors of shape*[batch_dim, input_length]*, embeds them into a new tensor of shape *[batch_dim, input_length, embed_size]*, and processes it through two fully connected layers that finally output a tensor of shape *[batch_dim, input_length, dictionary_size]*. Batch normalization and Layernorm are used in order to avoid overfitting. Adam has been used as optimizer and ReLU as activation function.

We used the Cross Entropy Loss in order to evaluate the quality of the classification performed. Despite our tries, the model seems to get stuck in a local minimum, that provides a relatively small edit distance and loss while mode collapsing into very poor results (a visualization which compares them with the next approach can be seen in the next paragraph). Anyway, here are the results in terms of average edit distance:

Table 2: Average edit distance on different datasets

Model	**Short Sentences**	**Randomized Short Sentences**	**Medium-length Sentences**
MLP	41.78 => 24.03% accuracy	42.15 => 23.36% accuracy	76.46 => 21.17% accuracy

2.1 Self Attention

A first modification that has come to our mind in order to try and improve the results before changing completely the infrastructure, was to introduce self attention into the model.

We are working with sequences, thus the model cannot obviously be invariant to permutations of the tokens. Therefore we inserted a learnable positional embedding layer before the self attention embedding.

The results given by the model seem to be pretty similar to the baseline in terms of edit distances. Nonetheless the sequences produced by this model show a huge improvement in terms of diversification of the output.

Here is a sample to show what we mean, and the results in terms of average edit distance:

Table 3: Sample to show difference

Gold	**FeedForward**	**FeedForward with Attention**
789c4b4ecc53c8cecb2f5728c948 - 2c011299c5003eda06bb	28282ccc282c2cccccc82 - c28c8ccc82c28c8	789c2bcc282858c8ccc82 - 128c8c8c8512cc8

Table 4: Average edit distance on different datasets

Model	**Short Sentences**	**Randomized Short Sentences**	**Medium-length Sentences**
SelfAttention	34.31 => 37.61% accuracy	38.03 => 30.85% accuracy	71.15 => 26.64% accuracy

3 Conv1D

The next step has been to give a one-dimensional Convolutional model a try. The idea comes from the fact that possibily the most significant relationships between the tokens were between adjacent tokens. We experimented with several setups, with results that aren't too dissimilar from the ones we obtained with the MLP and the MLP with self attention.

It's worth mentioning that we found out that inverting sequence length and embedding dimension brings the model to unexpected and astonishing results, even though the input data are in a wrong order. Follows a table that shows both the edit distances achieved with the wrong approach and with the right approach.

Table 5: Average edit distance on different datasets

Model	**Short Sentences**	**Randomized Short Sentences**	**Medium-length Sentences**
Conv1d	36.53 => 33.58% accuracy	39.156 => 28.80% accuracy	67.53 => 30.38% accuracy
Wrong Conv1d	5.98 => 89.12% accuracy	6.64 => 87.92% accuracy	29.90 => 69.17% accuracy

4 Recurrent Models

Recurrent Neural Network have been created to deal with sequences of values, therefore trying them for this task was very natural. There are many examples in the literature of RNNs that are able to predict correct values after long sequences of tokens. For our task we created an encoder-decoder architecture that encodes the whole input sequence and only after that starts decoding the hidden state generating the output sequence. The code is modular, making it possible to change between a simple RNN, LSTM and GRU with just one parameter in the first code block.

Each token of the input sequence is embedded with a standard pytorch Embedding layer before being passed to the encoder model. The output of the encoder model is not relevant to us so we just feed the decoder model with the hidden state of the encoder together with the 'START_OF_SEQUENCE' token, in order to kickstart the decoding process. At training time we used a *teacher forcing ratio* of 0.5: this means that 50% of the times the decoder is fed with the token that it predicted, while the correct token is given to it the other times.

Table 6: Average edit distance on different datasets by recurrent models

Model	**Short Sentences**	**Randomized Short Sentences**	**Medium-length Sentences**
RNN	20.868 => 62.05% accuracy	21.640 => 60.65% accuracy	-
GRU	8.5613 => 84.43% accuracy	8.6986 => 84.2% accuracy	-
LSTM	7.5913 => 86.19% accuracy	8.12933 => 85.21% accuracy	23.8973 => 75.37% accuracy

5 Seq2Seq with PLM

In recent years Pretrained Language Models (PLM) have gained a lot of attention due to their high capability to understand natural language and generate coherent and contextually relevant text, thereby revolutionizing various natural language processing tasks. Our task is a novelty in the field of NLP and we thought that leveraging the mentioned capabilities of PLMs could be a good idea, despite potential challenges related to the fact they are not used to generate apparently nonsensical strings and are in a sense already biased on generating meaningful sentences. Nevertheless it was worth trying training one of these models.

The HuggingFace library provides a lot of different pretrained models both encoder or decoder or both. Our task is essentially a sequence to sequence task, so we've used encoder-decoder models like Bart and T5. To help the model learn better when concluding a string we've added to the sentences in input a 'START_OF_STRING' token and a 'END_OF_STRING' token. The sentences are then tokenized with the pretrained tokenizer of the chosen model. We performed tests with the same dataset that we've used for the previous approaches.

Table 7: Average edit distance on different datasets

Model	**Short Sentences**	**Randomized Short Sentences**	**Medium-length Sentences**
BART-base	7.3 => 86.72% accuracy	8.9 => 83.81% accuracy	-
BART-large	6.9 => 87.45% accuracy	7.8 => 85.81% accuracy	29.86 => 69.28% accuracy
T5	11.2 => 79.63% accuracy	-	-

As shown in Table 1 BART-large obtain the smallest edit distance on the test set. This distance increases when tested on the randomized short sentences dataset, confirming the idea that the non-randomized short sentences dataset have a clear bias in the data. Given the promising results we experimented with different hyperparameters in this setup, facilitated by the wandb visualizations.

6 Finetuning an LLM: a prompt engineering approach

As last approach we decided to experiment with the cutting edge in the NLP field: Large Language Models. As mentioned previously in our discussion on Seq2Seq models, we opted for LLMs due to their significant power, although we were aware that this task might not diverge greatly from their training experiences. Therefore we did not anticipate superior performance compared to earlier models.

LLM training is computationally expensive due to their high number of parameters and impossible to train on Colab. That's why we used LoRa, which allowed us not to train all parameters of the model but just two smaller matrices that only after the training phase are merged with the original LLM. Again due to colab restrictions reasons we couldn't load a huge model, thus we decided to use a relatively small LLM called *facebook/opt-350m*. This model has 350 millions parameters, but thanks to PEFT (that supports QLoRa natively) and BitsAndBytes we loaded it into memory in 4bit quantization.

At test time we had fun experimenting with a prompt engineering approach, giving the model lots of different prompts to see how the results changed. Here's a couple of them:

1. *You are ChatGPT4. This is a string that must be encoded in a novel way that you must learn. Write down the encoded input string.*

2. *Take a breath and think step by step. Below is an input text that must be encoded in a novel way that has never been used before. Write a response that appropriately completes the request.*

This approach in the end showed some results but not as much to be compared to the previous models. In fact the outputs of this LLM were usually not consistent and very different from the targets. The model did learn to produce outputs in hexadecimal format, yet it struggled with consistency and occasionally generated random strings. We think that those performances are justified by the fact that LLMs are not created with apparently random seq2seq tasks in mind. This kind of task would have probably required a more powerful model or a much more aggressive setup in terms of trainable LoRA weights in order to give some appreciable results.

7 Non-working approaches: Bart

Before landing to an hex2hex architecture, we experimented with various configurations, many of which did not yield satisfactory outcomes. Let's go over one of the tries that were ultimately discarded as deemed inconclusive or insignificant.

One of the first approaches to the MLP baseline that we tried, before settling on one-hot encoding, involved using a pretrained tokenizer. The idea was to create a model capable of directly reading ASCII text without performing any kind of preprocessing, to tokenize and embed it, perform some calculations, and then output BART tokens, with the intention of successively detokenizing the prediction. We chose the BART tokenizer for this purpose. Given BART's training on natural language datasets, its tokenizer inherently focuses on linguistic constructs, which posed complications when applied to our unique dataset.

In our experimental observations, the model's predictions exhibited tokens that, on the surface, seemed to bear minimal difference from the target tokens, suggesting a promising direction. However, the subsequent detokenization phase highlighted significant discrepancies. Tokens that were very similar were detokenized into entirely different entities, often being translated into natural language words that bore no resemblance to the hexadecimal data we were analyzing. This divergence highlighted that, at least in our understanding of the model, the BART tokenizer was not the right path, persuading us to pivot.

A subsequent attempt involved using the BART tokenizer again, but this time only for the input, making the model output binary code instead of BART tokens. This binary output was then compared with the zipped version of the text files, which were read in binary rather than hexadecimal format. This approach was not as inconclusive as the previous one but nonetheless yielded poor results, possibly due to the length of the binary representation of the zipped files, which inevitably led to more error-prone predictions by the model. This led us to consider base64 encoding, and subsequently, hex encoding.

8 Computational Considerations and conclusions

This chapter tries to get a peak at how our models compare it against traditional zipping methods in terms of performance, resource consumption, and scalability.

As stated in the introduction, deterministic zipping applications leverage algorithms that are specifically designed for data compression, resulting in highly optimized applications in terms of speed and efficiency and ensuring a fast and lossless process. In contrast, neural network models (especially state-of-the-art ones like BART and T5 with large number of parameters), while showing promise in emulating the compression process, are resource-intensive, requiring substantial CPU and GPU power and possibly higher computation times.

Scalability is another crucial factor in the practical application of text compression techniques. Deterministic zipping applications are indeed capable of handling files of various sizes efficiently. The performance of Neural network models, instead, tends to degrade as the input sequence length increases both in terms of time and in terms of quality of the predictions.

To get some real data we decided to run a test: we compressed different txt files both with the Bart-large model (our best performing model) and with the traditional Deflate algorithm. What follows are the results merely in terms of time (ignoring for the moment the accuracy). The results of the Deflate algorithm were approximated, as the time windows were so short that there was a high variance in the measurements.

Table 8: BartLarge / Deflate comparison

Sentences	Bart-large model	Deflate
20 words sentence	3.74s	~10^{-3} s
150 words sentence	5.21s	~10^{-3} s

As can be easily seen from this test, considering the computational efficiency, resource usage, and scalability, deterministic zipping applications currently hold a practical advantage over neural network-based models for text compression. However nothing prevents neural network models in the future to complement or even enhance traditional compression techniques.