NEURAL NETWORK-BASED CHARACTER-TO-SYMBOL SEQUENCE TRANSLATION FOR TEXT COMPRESSION

Giuseppe Prisco
Sapienza University of Rome
prisco.1895709@studenti.uniroma1.it

ABSTRACT

The goal of the project is to to develop an innovative neural network model that can convert text into a format that is compatible with decompression software, thus providing an alternative to the process of file compression. The model is designed to translate human-readable text into a machine-readable symbol sequence that is compatible with unzip software. Although similar to [1], the task involves creating a model that accepts a text file as input and outputs a zip file, or the contents of the zipped file, similar to the output of zipping tools like 7-Zip. To assess the model's effectiveness, two metrics have been employed: the Levenshtein distance, which quantifies the similarity between the file created by the model and the one generated by the zipping software, and the 'unzip metric', which assigns a score of 1 if the selected software can successfully decompress the output file produced by the model, and 0 if it cannot. Ultimately, the aim of the project is to provide an alternative way of translating text into machine-readable symbols, making it easier to integrate with existing unzip tools and potentially revolutionizing the field of text compression.

Keywords Neural Network Model · Text-to-Symbol Translation · File Extraction

1 Introduction

Given the innovative nature of this project, I explored various approaches related to both the representation of input and output data formats, as well as alternative architectures to accomplish the task. Due to the runtime constraints imposed by Colab, such as RAM and GPU usage, I investigated which approach would work best within these limitations and could be successfully implemented on more powerful machines. The document begins with Section 2, which examines the creation of the dataset, pre-processing, and post-processing of data. It then introduces the baseline implemented in the project in Section 3, which is used for comparison with the final architecture to discuss the results obtained. Section 4 presents two different alternatives for the final model architecture, while Section 5 showcases the results in the form of plots for easier comparison. A Conclusion is provided at the end of the report. To maintain a concise document, all other intricate details have been included directly in the Colab notebook, where they are appropriately commented and analyzed.

2 Dataset Creation and Data Processing

For the chosen task we had the flexibility to create any text dataset we would like and generate the zipped version for each text. Given that the goal of the project is to provide a way of translating texts into their zipped version, I decided to load a dataset directly from the Hugging Face Hub[1] since it offers a wide variety of ready-to-use datasets for NLP tasks, including machine translation. The "GLUE" dataset[2] I selected already had "`train`", "`validation`" and "`test`" splits. For each of these splits, the texts were first **pre-processed** to obtain a pair of strings, the first representing the original text and the second representing its zipped version. This pair of strings was then encoded using the chosen tokenizer (either `T5` or `byT5`) and the final structure of the features consists of the encoded original texts (`input_ids`),

[1] Huggin Face Hub: `https://huggingface.co/docs/hub/index`
[2] GLUE: `https://huggingface.co/datasets/glue`

the attention mask for the `input_ids` (`attention_mask`) and the encoded compressed texts (`labels`). Among the various methods to produce the zipped version of each text, I decided to compare the zipped text produced by `zlib` with the zipped text produced by `zlib` and then encoded in `base64`. The aim was to evaluate the model's effectiveness in learning to translate different sequence formats and determine the best one. Finally, a **post-process** phase has been applied to the decoded versions of the string pairs to recover special characters that the tokenizer could not handle. As a final note, in some cases, the original dataset was processed to contain only a portion of the texts (around one third) to reduce the number of batches and to make the training process faster. Furthermore, the original texts were truncated to allow the model to be trained under Colab limitations and also "shortened" before truncation, as the compressed counterpart of short texts often contains more characters than the original version. In the notebook, there are two different versions of the data modules used: "`TextCompressionDataModule`" and "`MyBaselineDataModule`". Both function identically, but in the former, the Data Loaders are instantiated directly inside the module, while in the latter, they are explicitly defined outside of it.

3 Implementation of the Baseline

The baseline implemented in this project, designed as a reference for comparison with the final model architecture, was structured as a general model for sequence prediction tasks. Specifically, the "`MyBaseline`" model comprises three main components: an embedding layer, a GRU (Gated Recurrent Unit) layer, and a fully connected layer. The embedding layer is used to transform the input tokens into dense vectors of fixed size, known as embeddings, to capture more information about the tokens and their relationships with each other. The GRU layer, which is a type of recurrent neural network (RNN), processes these embeddings sequentially, capturing the dependencies between the tokens in the input sequence. The fully connected layer then maps the GRU outputs to logits for each token in the vocabulary, which are are then passed through a softmax function to compute the probabilities of the next token in the sequence. During training, the model is fed the original text and tasked with predicting the compressed text. In each of the training, validation, and test steps, the model computes the loss between the predicted token probabilities and the true next tokens.

The model also computes additional metrics, including the Levenshtein distance between the predicted and true sequences, and the success rate of unzipping the predicted and true sequences. These metrics will be analyzed later in a dedicated subsection. Despite achieving acceptable results for the Levenshtein distance (as low as 60%, indicating that the pair of strings has a distance equal to 60% of the maximum distance), the model seemed to only recognize common patterns of the compressed texts, in particular the characters at the start of the sequence[3].

Regarding other baseline alternatives, I considered using a different type of RNN, such as a LSTM (Long Short-Term Memory) network. However, due to its more complex internal structure compared to a GRU network (having three gates and a cell state, in contrast to the GRU's two gates), and the consequent requirement for more computational resources and memory, I decided to implement only the first baseline described.

4 Final Model Architecture

For the final version of the model, "`TextCompressionModel`", I opted to use the `T5` model from the Hugging Face Transformers library, which is a powerful model for sequence-to-sequence tasks. `T5` is an encoder-decoder model pre-trained on a variety of NLP tasks and for which each task is converted into a text-to-text format. Specifically, for my project, I used the "`T5ForConditionalGeneration`" model, which differs from the standard "`T5Model`" model by including an additional linear layer at the top of the decoder.

4.1 Model Training

Although `T5` has been pretrained to work well with translation tasks, since it was envisioned to perform translation from a language to another one (and not from human-readable texts to machine-readable symbols), I decided to retrain the whole architecture on my created dataset (Section 2).

To better evaluate the model's performance, I attempted to train my model with both "`t5_base`" and "`byt5_base`". The latter is a `T5` model pre-trained on byte sequences and could potentially perform better in my task since we are operating on character-to-byte translation rather than word-to-word translation.

The model is trained using teacher forcing, meaning that for the training we always need an input sequence and a corresponding target sequence pair, and dropout, used to prevent overfitting. The input sequence is fed to the

[3] Usually, the compressed sequence starts with the characters x\x9c ...

model using "input_ids" (corresponding to the tokenized original texts) and the target sequence, represented by "labels" (corresponding to the tokenized compressed texts) is shifted to the right. Additionally, the model is fed an "attention_mask" to differentiate between tokens that are actual words versus others that just represent the padding. The model outputs a probability vector, of size equal to the vocab_size of the model used[4], for each element of the input sequence and for each sequence in the batch. The result is that we have a shape for the logits in the form (batch_size, sequence_length, num_tokens). In my case (assuming t5 has been used), with a batch size of 8 and the sequence truncated or padded to 256 tokens, the shape is (8, 256, 32128).

The final model outperformed the baseline in all tested settings. Specifically, it demonstrated a better recognition of the format of the compressed texts, not merely recognizing the common patterns at the start/end of the output sequences. The results are presented in the following Section 5.

4.2 Hyperparameter search

To select the optimal hyperparameters for the model, I logged any significant result with a wandb logger[5]. Apart from train, validation and test losses and the metrics I implemented in the project, a handful of parameters are saved on the dashboard, ranging from total steps/epochs, batch sizes, the chosen model or encoding for the current run, the learning rate, dropout probability and more. Thus, the final configuration for the hyperparameters was chosen to be: a max sequence length of 256 for t5 and 64 for byt5, a number of encoder/decoder layers of 12 for t5 and 14 for byt5, a dropout of 0.1, and a learning rate of 0.00001. Other values were chosen to ensure the model could successfully run under Colab limitations, including a batch size of 8, a dataset truncation to the first 1000 training texts out of 2400, and a truncation of each string to the first 40 characters.

5 Metrics Employed and Comparison of Results

In this final section I present the results that have been obtained by training the models under various configurations, primarily between the use of a base64 encoding or not and the selection between t5 or byt5. However, before presenting the results, a more detailed paragraph is required to explain the metrics used to evaluate the models.

5.1 Implemented Metrics

For the task of making the model learn to translate from texts to their compressed version, two metrics were implemented in the project: the Levenshtein distance and the "unzip metric".

The **Levenshtein distance** is a string metric used to measure the difference between two sequences. It represents the minimum number of single-character edits required to change one sequence into another. However, the Levenshtein distance in its original form does not directly provide a representative result since we do not have a way of determining if a distance of, for instance, 50 is actually good or not. If the two sequences are very long, it can indeed be deemed as a good result. On the other hand, for very short sequences, it actually indicates that they have a large proportion of single-characters edits. Thus, I decided to not only compute the Levenshtein distance between the predicted sequence and the true sequence but also to compute the predicted sequence length and compare it to the true sequence. This allowed me to find the minimum and maximum possible Levenshtein distances (based of course on the length of the two sequences, in which the minimum distance is the difference of the two lengths and the maximum distance is the length of the longest sequence) and most importantly, the percentage Levenshtein distance. This last metric should be the one which clearly makes us understand the model's effectiveness since it computes the distance based on the length of the two sequences. Going into details, a percentage Levenshtein distance of 0% means that the two sequences are identical while a distance of 100% means that the two sequence differ in their entirety. In the following section, the best model I trained obtained a percentage Levenshtein distance of around 37%.

On the other hand, the "**unzip metric**" represents how many predicted sequences can successfully be decompressed. This metric assigns a score of 1 if the zlib library can successfully decompress the model's output sequence, and 0 if it cannot. Unfortunately, none of the trained models were able to successfully decompress any predicted sequence. This is primarily due to the fact that most zipping tools append a checksum of the string at the end of the file and specific headers at the beginning. For the model to decompress these sequences, it would need to accurately predict the entire compressed sequence, including the checksum and headers. This implies that the model should have a Levenshtein distance of exactly 0 to decompress the sequence, essentially requiring it to perfectly predict the compressed sequences for each possible original sequence.

[4] "vocab_size" is 32128 for t5 and 384 for byt5

[5] Weight and Biases: https://wandb.ai/site

5.2 Results

In this section I compared the results obtained with the different models and setups by showing the plots obtained for the relevant metrics and summarizing the results in Table 1. As shown in Figures 1 and 2, 3 and 4, the "`MyBaseline`" model fails to obtain decent results in both the Loss and Percentage Levenshtein Distance, with a validation loss that starts growing after a while and with the distance remaining pretty much stationary. As stated before, the model fails to recognize the overall structure of the compressed file, while just memorizing the starting structure.

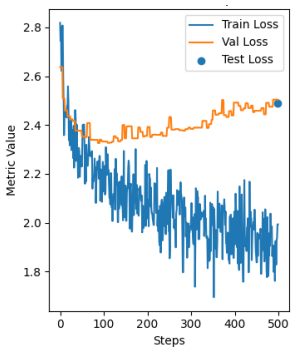

Figure 1: Losses for MyBaseline, byt5, No Encoding

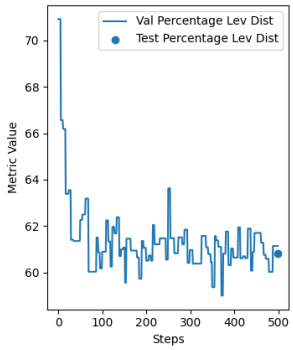

Figure 2: Percentage Levenshtein Distance for MyBaseline, byt5, No Encoding

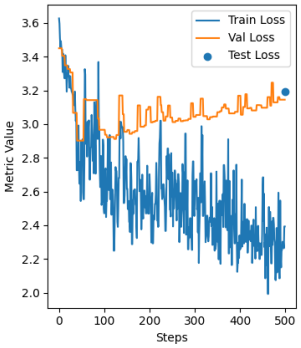

Figure 3: Losses for MyBaseline, byt5, Base 64 Encoded

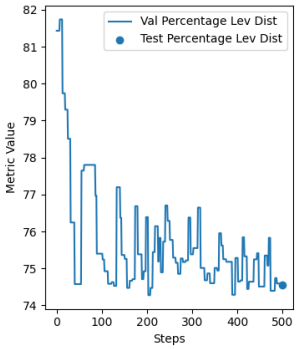

Figure 4: Percentage Levenshtein Distance for MyBaseline, byt5, Base 64 Encoded

Conversely, Figures 5-8 demonstrate the superiority of "`TextCompressionModel`", achieving much more stable losses with smaller values and excellent results for the Percentage Levenshtein Distance (with the most significant gains obtained when no encoding is used). As shown in the contents of the txt files[6] present in the Colab notebook, this final model is capable of recognizing the correct structure of the compressed strings and is able to mimic the behaviour of a zipping tool far more effectively than the baseline.

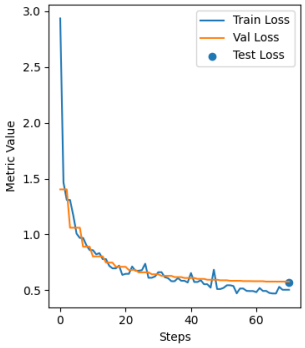

Figure 5: Losses for TextCompressionModel, t5, No Encoding

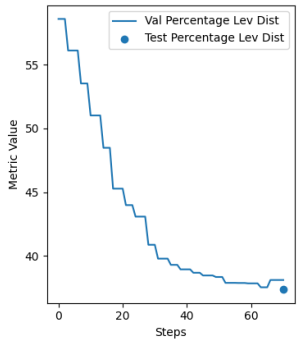

Figure 6: Percentage Levenshtein Distance for TextCompressionModel, t5, No Encoding

[6]The txt files created in the notebook contain the predicted sequence and the true sequence for better comparison

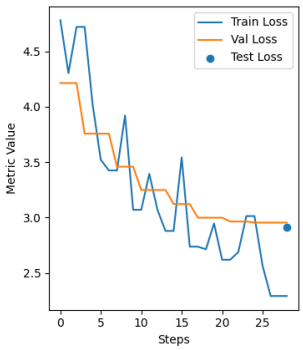

Figure 7: Losses for TextCompressionModel, byt5, Base 64 Encoded

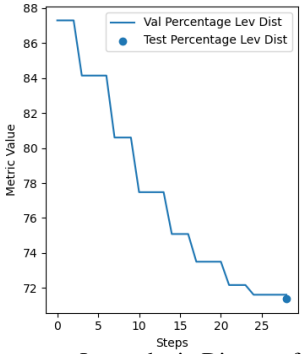

Figure 8: Percentage Levenshtein Distance for TextCompressionModel, byt5, Base 64 Encoded

To ensure the reproducibility of these results each time the notebook is executed, I selected a seed for the random numbers generated and initialized the Trainer with the "deterministic" flag set to "True". Ultimately, it was evident that the models trained under these different setups achieved superior results when no encoding was applied after the text compression phase.

In Table 1, I have summarized the results obtained by the various models.

Table 1: Models Validation/Test scores for the adopted Metrics

Model Names	Levenshtein Distance (in %)		Unzip Metric	
	No Encoding	Base 64	No Encoding	Base 64
MyBaseline (t5)	68.023	87.781	0	0
MyBaseline (byt5)	60.818	74.563	0	0
TextCompressionModel (t5)	37.347	71.688	0	0
TextCompressionModel (byt5)	37.154	71.382	0	0

For additional plots and other logged metrics, please refer to the Colab notebook.

6 Conclusion

In this report, I presented a potential approach to address the challenging task of translating human-readable text into machine-readable symbols. I compared the simpler "`MyBaseline`" model, based on GRUs, with the final architecture of the "`TextCompressionModel`" model, which utilized different versions of `T5`. As demonstrated in the "Results" section, the final model significantly outperforms the baseline for the Percentage Levenshtein Distance metric, achieving the lowest score of 37%. Unfortunately, none of the trained models could successfully decompress the predicted text, even with manually appending special headers and accurately formatted checksums to the sequence. Inherently, the model would need to achieve a distance of exactly 0% in order to correctly mimic the behaviour of the unzipping tools, a result that was beyond the scope of this project. Nevertheless, the model demonstrated its ability to recognize the patterns of any compressed string, hinting at future possibilities to successfully perform the given task. In conclusion, the prospect of having a model that could successfully compress and decompress any given text would provide significant efficiency in text compression, serving as an alternative to using zipping tools that require additional operations to compress and decompress text.

References

[1] Chandra Shekhara Kaushik Valmeekam, Krishna Narayanan, Dileep Kalathil, Jean-Francois Chamberland, and Srinivas Shakkottai. Llmzip: Lossless text compression using large language models. *arXiv:2306.04050*, 2023.