Adaptive Huffman and Recurrent Neural Network Combination for Text Lossless Compression

Dror Meidan, Ido Springer

July 2020

Abstract

Often, when a compression program sees words like "he is", it never automatically determines that a word like "going" is much more likely to follow a word like "be", and the reason for this is that the compression algorithm sees bits, and has no prior knowledge regarding which words follow which other words. In this work, we present a data lossless compression method based on language models (using a recurrent neural network), and uses the model output probabilities to build a different Huffman tree for each character in the file. Alongside implementation issues, we compare the suggested model which comes from the context mixing family to ZIP compression method (basically the Lempel-Ziv algorithm with other features) with a text file containing all of Charles Dickens writings.

1 Theoretical Background

1.1 Huffman Coding

There are several simple ways to compress a text file. Let's say we would like to compress the string `aabdcbab`. Without any compressing, each character symbol in the file is stored as a byte (8 bits, each bit is 0 or 1) according to the ASCII encoding. Therefore the size of the file in bytes is the number of characters in it. In this example, the encoding of our string will be

01100001	01100001	01100010	01100100	01100011	01100010	01100001	01100010

One byte can represent up to $2^8 = 256$ unique characters. In most cases, the file will not contain every ASCII character, so we do not need to save a whole byte for every character in the text. We only need to encode the characters that appear in the text. Therefore we can define a similar yet limited binary encoding to every character. In last example, we can define a code $\mathbf{c}$ by $\mathbf{c}(a) = 00$, $\mathbf{c}(b) = 01$, $\mathbf{c}(c) = 10$, $\mathbf{c}(d) = 11$, and then our string will be encoded as the concatenation

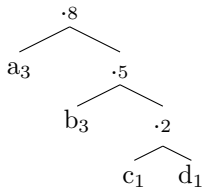
00	00	01	11	10	01	00	01

In this method, the size of every codeword is fixed. We could drop this requirement, and define a variable-length encoding. To guarantee that our code is uniquely decodable, we demand that the code will be a prefix-code, meaning there are no two code words that one is the prefix of the other (in other words, when we read the concatenated encoding, we will always know where does a single character encoding ends). For our string we can define the prefix code $\mathbf{c}(a) = 0$, $\mathbf{c}(b) = 10$, $\mathbf{c}(c) = 110$, $\mathbf{c}(a) = 111$, and get the encoding

0	0	10	111	110	10	0	10

The concept of variable-length encoding could lead us to better compression. We would like that frequent characters will have shorter coding, and the longer code words will be used to encode rare characters.

Huffman coding [1] gives us a prefix code with that property. First, we count all appearances of each character in the text. Then we build a tree based on the frequencies - starting with the characters as isolated leaves, at every stage we combine the two trees with the lowest total frequency until we get a single binary tree. This tree defines a prefix code. In our example, the tree will be



and the prefix code we get is the same as before.

As we see, the Huffman code computes the global frequency of every character. In consequence, it is optimal only where we have no context information. However, the original Huffman coding does not consider the dependencies between a character and its context. The frequency of every character might change in different environments. For instance, after 'The blue bo' we are more likely to observe an 'x', even though the global frequency of 'x' is usually low.

1.2 Language Models

A *language model* is a probability distribution over a language, meaning it assigns a probability $P(w_1, ..., w_n)$ to every sequence of words $w_1, ..., w_n$ (characters, in our case). Such probability function can be used to get the distribution of a word based on its context, $P(w_1, ..., w_{n-1}|w_n)$. The context can be defined in several ways, here we define the context as the previous n words (otherwise we are looking at the future, and we will not be able to decompress).

Language models are used in a wide range of Natural Language Processing (NLP) tasks. Since the text file were given contains a natural language text, we thought we could use this prior with our knowledge from NLP.

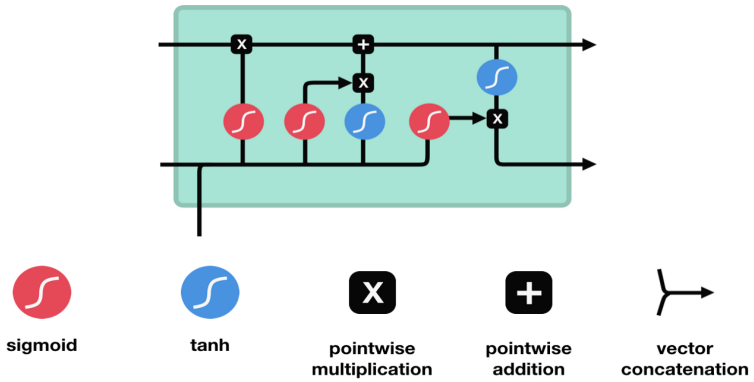


Figure 1: LSTM Cell and Its Operations

Estimating the probability of a sequence of words is not an easy task. There are a lot of algorithms to train a language model. Some algorithms are based on counting and statistics, while more complex algorithms are based on machine learning and in recent years deep learning. For instance, The Hidden Markov Model (HMM) assumes that the word context is quite narrow (only 2-3 previous words), and then by counting n-grams of words in our data we can compute a probability score for every sequence of words. The problem with that attitude is that sometimes we would like to consider a wider context, and we cannot rely on counting alone. Firstly, because every n-gram becomes rare when n is large, and secondly because saving all n-grams counts will cost a lot of memory.

1.3 Long Short-Term Memory

One common way of training a language model is using Recurrent Neural Networks (RNN) and in particular Long short-term memory (LSTM) deep neural networks. LSTM is a common RNN type designed with sophisticated architecture to avoid the vanishing gradient problem. A common architecture is composed of a cell (the memory part of the LSTM unit) and three "regulators", usually called gates, of the flow of information inside the LSTM unit: an input gate, an output gate and a forget gate.

Intuitively, the cell is responsible for keeping track of the dependencies between the elements in the input sequence. The input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit. The activation function of the LSTM gates is often the logistic sigmoid function.

Figure 1 shows an illustration of LSTM cell and its operations. In this work, we use LSTM for character-level language modeling.

2 Adaptive Huffman with LSTM

2.1 The Algorithm

Our suggestion combines the Huffman coding and the LSTM language model using adaptive Huffman coding. Practically, for each character in the file, one Huffman tree is calculated according to the prediction of a character-level LSTM language model, given the character context. Another tree is the fixed global character distribution Huffman tree. For each character we decide which tree to use according to a threshold. As a consequence, the method gets the benefits from the short coding of the Hoffman tree and the context information from the LSTM outputs.

Algorithm 1: Compression

```
Train a character-level LSTM model;
Compute fixed Huffman coding;
Initialize context and binary text;
for character in file do
    probabilities = Softmax(LSTM(context)));
    Build adaptive Huffman tree using the probabilities;
    Apply threshold to prune the tree;
    if character leaf survived the threshold then
        text = text + '0' + leaf path in adaptive Huffman tree
    else
        text = text + '1' + leaf path in fixed Huffman tree
    end
    Update context;
end
Save LSTM parameters in file;
Save compressed text with the relevant dictionaries;
```

The decompression process is similar. We keep a flag that remembers if the current bit-stream we are trying to decode starts with 1 or 0. In this way we

know if to use the context based Huffman tree or the fixed Huffman tree.

Algorithm 2: Decompression

```
Load LSTM parameters;
Load compressed binary text and the relevant dictionaries;
Re-build fixed Huffman tree using the character count dictionary;
Initialize context, decompressed text, current coding and adaptive flag;
Compute initial adaptive Huffman tree;
for bit in compressed text do
    if flag = 0 then
        current code = current code + bit;
        if current code is a leaf in the current adaptive Huffman tree
         then
            Decode next character;
            Update context;
            Re-compute adaptive Huffman tree;
            Reset current code;
        end
        continue;
    else if flag = 1 then
        current code = current code + bit;
        if current code is a leaf in the fixed Huffman tree then
            Decode next character;
            Update context;
            Re-compute adaptive Huffman tree;
            Reset current code;
        end
        continue;
    else
        flag = bit
    end
end
Save decompressed text;
```

2.2 Language Model Architecture

Although there are several differences in the neural network language models that have been successfully applied so far, all of them share some basic principles [2]: First, The input characters are encoded by a learned embedding matrix. Second, At the output layer, a softmax activation function is used to produce correctly normalized probability values.

The training process of the LSTM model is similar to other methods that use neural networks [3, 4]: the previous ten characters (the context) are entered to the LSTM which tries to predict the next character. Then, we compare the

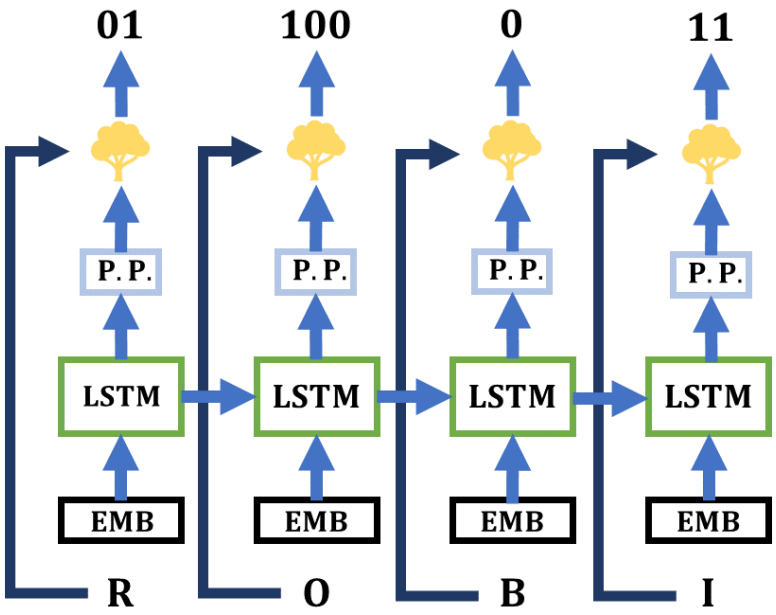


Figure 2: Illustration of the algorithm and the language model. In the presented example the model compresses the word "ROBI". The first layer embeds each character to vector (the embedding layer). Then, each LSTM cell gets the embed vector and vector from the previous LSTM cells and outputs the prediction probabilities to the next character. With these probabilities, the algorithm builds the adaptive Huffman tree, and with this tree, it codes the character to a binary sequence.

predicted character to the real character and calculate the loss between them. Finally, the model updates its parameters according to the loss with gradient-descent based method. Specifically, we use Adam optimizer. This procedure is repeated until it goes over all of the text for one epoch.

The architecture of the language model is very basic: The first layer takes the character and embed it to vector in size of 200 (embedding matrix). Then the vector enters the LSTM layer which out a vector of size of the vocabulary. In the training phase, we take this vector to calculate the loss, and in the compress/decompress phase the Huffman tree is built according to those probabilities (after applying softmax). The LSTM has two layers with 200 hidden units. An Illustration of the algorithm and the language model is presented in Figure 2.

2.3 Pruning the Trees

One main problem of Huffman coding is highly imbalanced trees. The maximal Huffman code length of one character taken from a vocabulary of size n is theoretically $n-1$. Usually, the Huffman tree is more balanced, so the code length is closer to $log(n)$. In adaptive Huffman coding, the trees tend to be less balanced. We assume this is because the adaptive tree is based on the local context rather than the global character distribution.

To solve this problem, we suggest pruning the Huffman trees with a filter threshold. There are two types of thresholds that can be applied: a discrete one – the number of levels in the tree, or a continuous one – a constant threshold of the LSTM probabilities. At every step, we compute the Huffman tree of a character based on the LSTM output probabilities and then prune the tree with a threshold. If the character leaf survived the pruning, we encode it with a similar Huffman encoding (as the path in the binary tree). Otherwise, we encode it with the fixed global Huffman tree. We add the code a prefix to distinguish between characters that were encoded using the adaptive Huffman tree and characters that were encoded using the fixed Huffman tree.

2.4 Loss Function

At most cases of language models training, the training criterion is the cross-entropy error which is equivalent to the maximum likelihood. Although the choice of loss function looks arbitrary, for our purpose is critical. While in most machine learning tasks the maximum argument of the output is taken, in our task, we use all prediction probabilities to build the Huffman tree. As we discussed in the last subsection, we prefer the Hoffman tree to be balanced, therefore, we prefer the output probabilities will be more absolute and less smooth. This behavior of probabilities combines with trees pruning (especially with the continuous threshold). In this subsection, we look at some alternatives:

1. The cross-entropy loss function defined as:

$$L_{CE} = -\log(\hat{y}_i) \tag{1}$$

   where $\hat{y}_i$ is the prediction probability of the real word (i) of the sentence. This loss function cause to maximum entropy of the prediction probabilities, therefore, we will ask to find another loss function.

2. The L_2 loss function (Mean Squared Error) defined as:

$$L_{MSE} = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2}$$

   where $\hat{y}_i$ and y_i are defined as above. The MSE loss function prohibits the parameters of the model to be large but still has problems diminishing very small values.

3. The L_1 loss function (Mean Absolute Error) defined as:

$$L_{MAE} = \sum_{i=1}^{n} |\hat{y}_i - y_i| \tag{3}$$

where $\hat{y}_i$ and y_i are defined as above. The output probabilities under the MAE loss function are very sparse, resulting very high probability to the reasonable character and negligible probability to others. This behavior is preferred for building the Huffman tree.

2.5 Implementation Issues

As was asked, we take care of the non-alphanumeric characters. *Alphanumericals* are a combination of alphabetical and numerical characters and is used to describe the collection of Latin letters and Arabic digits or a text constructed from this collection. Our solution is to define a new character for every non-alphanumerical shown, therefore, our vocabulary includes also these characters (e.g. dot and comma, or non-ascii characters).

2.5.1 Initial Context Padding

At each step, we use the LSTM model to predict the Huffman tree based on the context. The context we use is the previous 10 characters (as a sliding window), therefore, there is a problem for the first 10 characters which lacking full context. We solve this by adding a special symbol for start, in this way the context of the first 10 characters is padded with the start symbol, so the context size is always the same. Notice, we need to add the start symbol also during the training of the LSTM model. Although LSTM as a recurrent network can handle input size of arbitrary length, we keep the input size similar. We use this initial context padding also in decompression.

2.5.2 End of File Symbol

Files are stored in memory as a list of bytes, meaning that the number of bits is a multiplication of 8. When we compress a text file, if we will not fill the last byte with 8 bits, the operating system will write some random bits at the end of the file to complete it. These random bits might hurt the decompression process. To solve this problem, we add another special symbol marking the end of the file. Practically, we pad this symbol with some zero bits so our compressed file is always the same. In decompression, we stop reading the compressed file after we see the end of the file symbol.

3 Results

threshold type	threshold	LSTM loss	compression size in bytes	compression ratio
no threshold	-	Cross-Entropy	10606	1.0606
continuous	0.02	Cross-Entropy	6441	0.6441
continuous	0.01	Cross-Entropy	6624	0.6624
continuous	0.001	Cross-Entropy	7428	0.7428
discrete	10	Cross-Entropy	6839	0.6839
discrete	20	Cross-Entropy	8398	0.8398
discrete	50	Cross-Entropy	10307	1.0307
continuous	0.01	L1	**6182**	**0.6182**
discrete	10	L1	6579	0.6579
discrete	20	L1	7158	0.7158
continuous	0.05	L2	6391	0.6590
continuous	0.01	L2	6590	0.6590
discrete	10	L2	6881	0.6881
discrete	20	L2	8216	0.8216

Our algorithm tends to be very slow (see discussion), so we cannot compress the whole Dickens writings file. We experimented the compression of the first 10000 characters in this file.

We compare some arguments of the algorithm: threshold type and LSTM loss function. Where we do not use any threshold, the adaptive Huffman tree hurts the compression rate because we give the weight only to the context, which gives us unbalanced trees. When we add the thresholds (discrete and continuous) the adaptive Hoffman tree is getting more balanced, hence the ratio getting better. Note that the thresholds help only where the adaptive tree is small. The table shows the continuous threshold is better but not by a big difference. The higher the threshold is, the better the algorithm compresses the file.

Another change that helps to get a better compression ratio is choosing a correct LSTM loss function. As we saw in the previous section, the L_1 loss function gives us more balanced adaptive trees. The benefit of this change is reflected in the table with a better compression ratio.

Our best compression ratio is 0.6182. The original file size is 31,457,486 bytes, so it would be compressed to about 19,447,017 bytes. After adding the LSTM model size and the dictionaries which have 2,751,755 bytes we would get a compressed file of size 22,198,772 bytes. ZIP compressed the same file to 11,556,090 bytes, so ZIP is better.

4 Discussion

In this project, we present a novel method for compressing text files. We show that the regular Huffman coding can be improved by focusing on the local character distributions. We account for the context of a given character by training a neural language model and use the model output probabilities of the context to build an adaptive Huffman tree. This method does not over-perform the ZIP algorithm, yet we suggest some possible improvements that might be implemented.

We model the language distribution using Long Short Term Memory (LSTM) [5]. The major advantage of this architecture is that it can handle variable length input. Another advantage is that LSTM can handle long input without the gradients will vanish during training. Therefore, we can use a wide context for each character. LSTM is not the only way to do it, and other works use a simple Feed-Forward Neural Networks or Transformers [4]. Since LSTM is a recurrent network, it has conceptually fewer parameters (because we pass through the same layer over and over), and for compression purposes, we believe it is preferred.

In this work, we only use the information of the given text file for training the language model. We also save the trained model parameters along with the binary compressed text in the same file. It is reasonable, assuming that any compression algorithm must use only the information of the given data to compress. However, if we remove that constraint, the language model can be trained using a larger external corpus. If we have a large corpus that has a similar distribution to the file we would like to compress (e.g. both files are in English), we can train the language model on the external corpus and use its parameters to compress the other file. In this way we can compress also small text files – the distribution of the file itself is not always enough, but using another corpus we can catch the prior distribution, and get better results compressing the small file. The next step would be using a pre-trained language model, a popular area of research in Natural Language Processing. Starting with basic pre-trained word (or character) embeddings such as Word2Vec [6] or more complex pre-trained language models such as BERT [7].

Our method suffers from two major problems: time complexity and model size. Notice that the time complexity is only crucial during the decompression. As for compression, we can use batching to save time. Since in compression time we always know the character context, we can compute in parallel the LSTM outputs and the adaptive Huffman trees for each context (we only have to save the compressed bits in the right order). However, during decompression, we must decode all previous characters to get a specific character context.

As for the LSTM model size, it can be reduced in various ways. The obvious way is tuning the LSTM hyperparameters (such as embedding layer size, number of layers, and layer size), and the model performance will not be significantly hurt. We did not explore many hyperparameters configurations in this work. The embedding matrix can also be reduced. Notice that the embedding matrix dimension is affected by the vocabulary size. Therefore when using word embeddings, the embedding matrix has many more parameters than when using character embeddings, and this is why we use a character context. Current NLP models use Byte-Pair encoding [8] to reduce the vocabulary size, this method can also be applied here. Recently, several interesting papers are dealing with neural network pruning, suggesting that we can remove the vast majority of the model parameters while keeping a similar performance [9].

We acknowledge Prof. Reuven Cohen for a fascinating and interesting Information Theory course and wish him a pleasant summer.

5 Code

Our implementation can be found here: `https://github.com/IdoSpringer/text_compression`, along with detailed the running instructions. We used the PyTorch package in python for the model training. It is recommended to run the software on a GPU device.

References

(1) D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

(2) M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

(3) A. El Daher and J. Connor, "Compression through language modeling," *NLP courses at Stanford, URL: http://nlp. stanford. edu/courses/cs224n/2006/fp/aeldaher-jconnor-1-report. p df (accessed on April 21st, 2016)*, 2016.

(4) F. Bellard, "Lossless data compression with neural networks," 2019.

(5) S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

(6) T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

(7) J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

(8) P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

(9) H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow," *arXiv preprint arXiv:2006.05467*, 2020.