Lossless Data Compression with Neural Networks

Fabrice Bellard

May 4, 2019

Abstract

We describe our implementation of a lossless data compressor using neural networks. We tuned Long Short-Term Memory and Transformer based models in order to achieve a fast training convergence. We evaluated the performance on the widely used `enwik8` Hutter Prize benchmark.

1 Introduction

Although the best lossless data compressors are based on neural network models [7], they use them mostly to mix the statistics from a large number of hand coded models. Here we present lossless data compressors using pure neural network models based on Long Short-Term Memory (LSTM) and Transformer models.

The lossless data compressor employs the traditional predictive approach: at each time t, the encoder uses the neural network model to compute the probability vector p of the next symbol value s_t knowing all the preceding symbols s_0 up to s_{t-1}. The actual symbol value s_t is encoded using an arithmetic encoder with approximately $-\log_2(p_{s_t})$ bits. Then the model is updated knowing the symbol s_t. The decoder works symmetrically so there is no need to transmit the model parameters. It implies both encoder and decoder update their model identically.

When no preprocessing is done, s_t represents the byte at position t. Hence there are $N_s = 256$ different symbol values from 0 to $N_s - 1$.

When preprocessing is employed, each symbol represents a sequence of input bytes. There is a vocabulary of about $N_s = 16000$ symbols in our larger models.

Since the input data is presented only once to the neural network, our compression results correspond to the training on a single epoch. Hence they show the speed of convergence of the various models. This information is usually not clearly shown in the published litterature and is useful to design better models or gradient optimizers.

For easier replication of the results, the source code of our models is available[1].

[1] `https://bellard.org/nncp`

2 Long Short-Term Memory Model

2.1 Model

The model consists in L layers of a LSTM cell. Each cell takes as input the output of the corresponding cells of the previous layers and the previous symbol.

For each layer $l = 1...L$, the cell is defined as:

$$
\begin{aligned}
f_{t,l} &= \text{sigm}(\mathit{LayerNorm}(W_l^f[h_{t-1,l}; h_{t,0}; ...; h_{t,l-1}])) && (1)\\
i_{t,l} &= \text{sigm}(\mathit{LayerNorm}(W_l^i[h_{t-1,l}; h_{t,0}; ...; h_{t,l-1}])) && (2)\\
o_{t,l} &= \text{sigm}(\mathit{LayerNorm}(W_l^o[h_{t-1,l}; h_{t,0}; ...; h_{t,l-1}])) && (3)\\
j_{t,l} &= \tanh(\mathit{LayerNorm}(W_l^j[h_{t-1,l}; h_{t,0}; ...; h_{t,l-1}])) && (4)\\
c_{t,l} &= f_{t,l} \odot c_{t-1,l} + \min(1 - f_{t,l}, i_{t,l}) \odot j_{t,l} && (5)\\
h_{t,l} &= o_{t,l} \odot c_{t,l} && (6)
\end{aligned}
$$

The input of the first layer is set such as

$$h_{t,0} = \mathit{One}(s_{t-1})$$

The probabilies p_t for the symbol at time t are computed as:

$$p_t = \text{softmax}(W^e[h_{t,1}; ...; h_{t,L}] + b^e)$$

W_l^f, W_l^i, W_l^o, W_l^j, W^e, b^e are learned parameters. $c_{t,l}$ and $h_{t,l}$ are vectors of size N_c. $h_{-1,l}$ and $c_{-1,l}$ are set to the zero vector.

$[a_0; ...; a_n]$ represents the concatenation of the vectors a_0 to a_n. $\odot$ is the element-wise multiplication.

$y = \mathit{One}(k)$ is a vector of size N_s such as $y_i = 0$ for $i \neq k$ and $y_k = 1$

$y = \mathit{LayerNorm}(x)$ is defined as:

$$
\begin{aligned}
y_i &= \frac{(x_i - \mu)}{\sigma + \epsilon} g_i + b_i \text{ with } i = 0 \dots N-1\\
\mu &= \frac{1}{N} \sum_{i=0}^{N-1} x_i\\
\sigma &= \sqrt{\frac{1}{N} \sum_{i=0}^{N-1} (x_i - \mu)^2}\\
\epsilon &= 10^{-5}
\end{aligned}
$$

where g and b are per instance learned parameters.

Model	L	N_c	N_s	Learning rate	#Params (total)	#Params (no embed)	Memory (bytes)
Small	3	90	256	0.007	542k	265k	12.1M
Large1	5	352	16388	0.006	151M	36.3M	1.43G
Large2	7	384	16388	0.006	224M	48.0M	2.04G

Table 1: LSTM model parameters

2.2 Model specificities

The model is close to the one described in [1]. We modified the cell equation (5) to use the cell output variant from [2]. This change is important because it ensures the cell state is bounded unlike in the conventional LSTM. Moreover, we removed the unnecessary tanh function in equation (6) because the cell state is already clamped between -1 and 1.

We added layer normalization operations (*LayerNorm*) as suggested in [3]. They give a noticeable gain in performance.

No dropout is used because overfitting is unlikey to happen as the samples are presented once to the model.

2.3 Training details

Regular truncated backpropagation is applied on successive segments of 20 timesteps. The initial states $c_{t,l}$ and $h_{t,l}$ of the training segment are set to the last values of the previous segment.

A small batch size B of 16 is used to have a fast training. Smaller batch sizes give slightly better performance but the matrix operations are less efficient, so it is a compromise.

We use the Adam optimizer [5] with $\beta_1 = 0$, $\beta_2 = 0.9999$ and [2] $\epsilon = 10^{-5}$. No gradient clipping is done.

2.4 Model parameters

In table 1 we provide the parameters of our LSTM models. We listed the number of parameters without the input embeddings because the input embeddings have a negligible computing cost. The "Memory" column gives the total memory allocated by our implementation to evaluate and train the model. The small model has a similar complexity as `lstm-compress` [8]. It allows to compare the speed of both implementations.

[2] In our implementation, ϵ is added to the running average before taking the square root.

3 Transformer Model

3.1 Model

Our model consists in L layers of a Transformer cell.

For each layer $l = 1...L$, for each attention head $i = 1...N_h$, for each backward time step $j = 0...M-1$:

$$
\begin{aligned}
q_{t,l,i} &= W^q_{l,i} h_{t,l-1} && (7)\\
k_{t,l,i,j} &= W^k_{l,i} h_{t-j,l-1} && (8)\\
v_{t,l,i,j} &= W^v_{l,i} h_{t-j,l-1} && (9)\\
a_{t,l,i,j} &= k^{\mathsf{T}}_{t,l,i,j}(q_{t,l,i} + u_i) + w^{\mathsf{T}}_{i,\min(j,d_{max})}(q_{t,l,i} + v_i) && (10)\\
r_{t,l,i} &= \mathrm{softmax}(\frac{a_{t,l,i}}{\sqrt{d_{key}}}) && (11)\\
u_{t,l,i} &= \sum_{k=0}^{M-1} r_{t,l,i,k}.v_{t,l,i,k} && (12)\\
o_{t,l} &= LayerNorm(W^p_l.[u_{t,l,1}; ...; u_{t,l,N_h}] + h_{t,l-1}) && (13)\\
e_{t,l} &= W^g_l.\mathrm{ReLU}(W^f_l o_{t,l} + b^f_l) && (14)\\
h_{t,l} &= LayerNorm(e_{t,l} + o_{t,l}) && (15)
\end{aligned}
$$

The input of the first layer is set such as:

$$h_{t,0} = W^{ei}.One(s_{t-1}) \ .$$

The probabilies p_t for the symbol at time t are computed as:

$$p_t = \mathrm{softmax}(W^{eo} h_{t,L} + b^{eo}) \ .$$

$h_{t,l}$ and $o_{t,l}$ have a dimension of d_{model}. $q_{t,l,i}$ and $k_{t,l,i}$ have a dimension of $d_{key} = \frac{d_{model}}{N_h}$. $v_{t,l,i}$ has a dimension of $d_{value} = d_{key}$. b^f_l has a dimension of d_{inner}.

$W^q_{l,i}$, $W^k_{l,i}$, $W^v_{l,i}$, u_i, $w_{i,j}$, v_i, W^p_l, W^f_l, b^f_l, W^g_l, W^{ei}, W^{eo}, b^{eo} are learned parameters.

3.2 Model specificities

The model is close to the one defined in [4]. Learned relative positional embeddings are used instead of sinusoidal relative embeddings. We did not evaluate the later, but learned positional embeddings are simpler and according to [6] give good

Model	L	d_{model}	d_{inner}	d_{max}	N_s	Learning rate	#Params (total)	#Params (no embed)	Memory (bytes)
Large	6	512	2048	64	16388	1e-4 $\rightarrow$ 3e-5	35.7M	27.3M	348M

Table 2: Transformer model parameters

results. We truncated the number of embeddings to d_{max} as increasing it does not significantly increases the performance (but decreases the convergence speed).

Untied embeddings gave better results in our tests.

No dropout is used because overfitting is unlikely to happen as the samples are presented once to the model.

3.3 Training

Truncated-like backpropagation is used on successive segments of 64 timesteps. $M = 128$ is used, so that 64 additional timesteps contribute to the attention computation (the $h_{t,l}$ values of these timesteps are reused as in [4]).

A small batch size B of 1 is used to have a faster training convergence.

We use the Adam optimizer[5] with $\beta_1 = 0$, $\beta_2 = 0.9999$ and $\epsilon = 10^{-5}$. No gradient clipping is done.

3.4 Parameters

In table 2 we provide the parameters we tested. There are $N_h = 8$ attention heads. The learning rate was linearly reduced. We summarize it by showing the initial and final values.

4 The NC library

In order to ensure that the model is exactly the same in the encoder and in the decoder, we developed a custom C library to implement the various operations needed by the models. It only depends on basic IEEE 754-2008 32 bit floating point operations (fused multiply-add, division, square root) with no reliance on external libraries to ensure we get deterministic results on every CPU and operating system.

On x86 CPUs, the AVX2 + FMA instructions are mandatory in order to get correct performance. The basic matrix operations were optimized to be fast with small matrixes. It happens in our case because we use small batch sizes. General purpose BLAS routines are much slower in these cases.

The library represents the forward evaluation of the model as a bytecode. The bytecode to compute the gradient using backward propagation is automatically

computed from this bytecode. Liveness analysis and dead code removal is done to reduce the memory usage and improve cache hits.

Since the decoder needs to evaluate the model incrementally at each timestep (but still does the training on a whole segment as the encoder), the library automatically converts the per-segment forward evaluation bytecode into a per-timestep bytecode. The resulting per-timestep forward evaluation bytecode usually runs slower because smaller matrix multiplications are involved.

5 Preprocessor

In order to improve the compression ratio and speed, we added a preprocessing step.

5.1 CMIX preprocessor

For the small LSTM model, we reused the text preprocessor of CMIX [7] and `lstm-compress` to have a meaningful comparison. This preprocessor replaces words from a prebuilt dictionnary of about 45000 words with a unique code from 1 to 3 bytes. All uppercase letters are converted to lowercase ones with escape codes.

5.2 Subword based preprocessor

The larger models use a preprocessor where each symbol represents a sequence of bytes.

A first pass is done over the input data to convert all uppercase letters to lowercase ones with escape codes as in the CMIX preprocessor. An escape code followed by a space is added before a word when it is not preceded by a space. These transforms help reducing the vocabulary size by having less variants for a given word.

The second pass iteratively adds a new symbol to the vocabulary by concatenating two well chosen symbols. It is inspirated from the Byte Pair Encoding [11]. The symbols to concatenate are chosen by maximizing the expected zeroth order entropy reduction. We also remove symbols when their frequency is below a given threshold. The second pass makes no assumption about word separators, so it works with any language.

For our models, we used a vocabulary of about 16000 symbols.

Program or model	Compr. Size (bytes)	Ratio (bpb)	Compr. Speed (kB/s)
`gzip -9`	36 445 248	2.92	17400
`xz -9` [9]	24 865 244	1.99	1020
`lstm-compress` [8]	20 494 577	1.64	4.4
CMIX (v17) [7]	14 877 373	1.19	1.56 †
LSTM (small)	20 500 039	1.64	41.7
Transformer	18 126 936	1.45	1.79
LSTM (large1)	16 981 765	1.36	3.25
LSTM (large2)	16 791 077	1.34	2.38

Table 3: Compression results for `enwik8`. † indicates that the figures comes from [10].

Program or model	Compr. Size (bytes)	Ratio (bpb)
`gzip -9`	322 591 995	2.58
`xz -9` [9]	197 331 816	1.58
CMIX (v17) [7]	116 394 271	0.93
Transformer	130 440 948	1.04
LSTM (large1)	128 292 351	1.03
LSTM (large2)	125 623 896	1.00

Table 4: Compression results for `enwik9`.

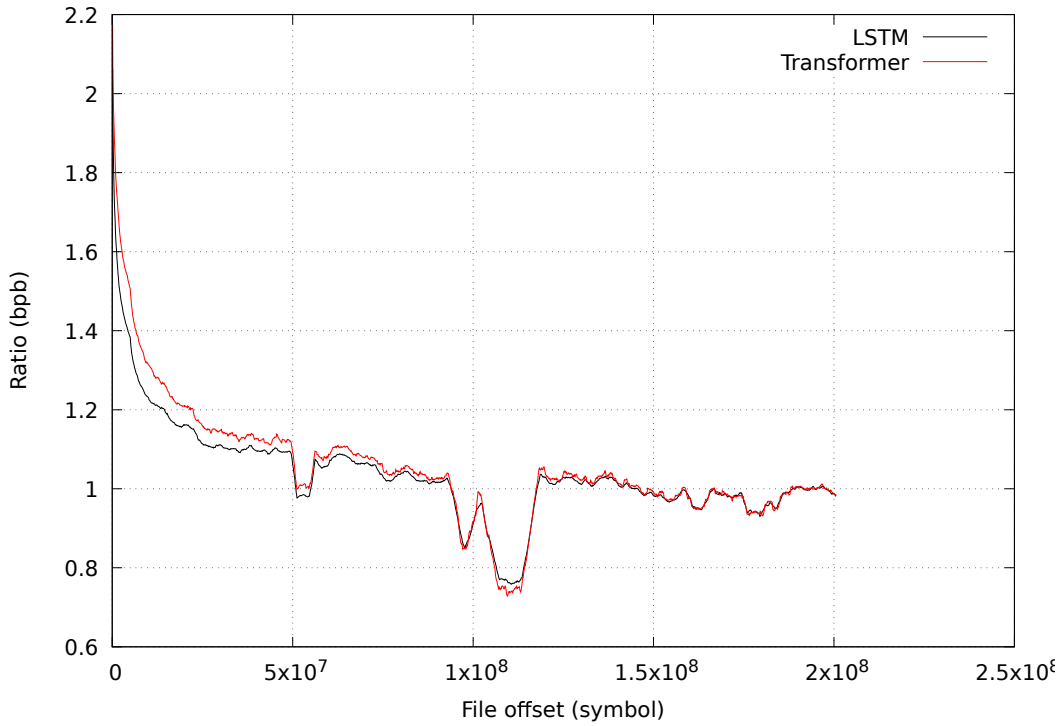

Figure 1: Compression ratio for the preprocessed `enwik9`. The ratio is computed with a moving average over 5MB.

6 Results

The results in bytes and bpb (bits per input byte) for the various models are given in table 3 and 4 for `enwik8` (first 100 MB of the English version of Wikipedia) and `enwik9` (first GB). The results for two popular compression programs are included. We show the results of CMIX [7], one of the best lossless compressor for this benchmark.

Note that these results cannot be directly compared with state of the art language modeling results such as [4] or [6] because:

- We do a single pass over the whole data (e.g. a single epoch).
- The result is the average bpb over the whole data instead of the last 5MB.
- The model parameters would have to be stored in the compressed file.

We did not take into account the size of the preprocessing dictionary in the compressed results because it is only 60 kB long when compressed with `xz` [9], which represents 0.005 bpb for `enwik8`. The size of the decompression programs is also small regarding the compressed output, so it is not taken into account in the results.

The compression speed is for a single core of a Core i5-4570 at 3.2 GHz except for the large models where 4 Xeon E5-2640 v3 cores at 2.6 GHz were used. The decompression speed is slower (1.5x slower for the LSTM model, 3x slower for the Transformer model) because the models must be evaluated timestep per timestep instead of on the whole training segment at once.

Regarding the speed, our results show that our implementation is much faster than `lstm-compress` [8] with a similar model and gain. The speed improvement comes from the more optimized matrix multiplication implementation and the larger batch size (16 instead of 1).

Regarding the compression ratio, we still do not reach the performance of CMIX (1.34 versus 1.19 bpb on `enwik8`) but we are among the best performers with a model which is much simpler.

In all our experiments, the Transformer model has a worse performance than the LSTM model although it gives the best performance in language modeling benchmarks ([4], [6]). The figure 1 shows the variation of the compression ratio with respect to the file offset for the large1 LSTM and Transformer models. It shows that our Transformer model has a lower convergence speed than the LSTM model, even if it ultimately performs similarly or better with a number of parameters in the same range (we exclude the input embedding parameters from our comparison because they have a negligible impact on the computation time). More experiments are needed to see if it is possible to do better with the Transformer model.

7 Conclusion

We presented a practical implementation of an LSTM based lossless compressor. Although it is slow, its description is simple and the memory consumption is reasonable compared to compressors giving a similar compression ratio. It would greatly benefit from a dedicated hardware implementation.

There are still many opportunities to improve the compression ratio, for example by using larger models, tuning the hyperparameters, using a better gradient optimizer or improving the preprocessor.

This experiment would not have been possible without our NC library which ensures deterministic evaluation and training with good performances on standard CPUs.

References

[1] Alex Graves, *Generating sequences with recurrent neural networks. CoRR, abs/1308.0850, 2013.*, arXiv preprint, arXiv:1308.0850, 2013.

[2] Gábor Melis, Chris Dyer, Phil Blunsom, *On the State of the Art of Evaluation in Neural Language Models*, arXiv preprint, arXiv:1707.05589, 2017.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton, *Layer normalization*, arXiv preprint, arXiv:1607.06450, 2016.

[4] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, Ruslan Salakhutdinov, *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*, arXiv preprint, arXiv:1901.02860, 2019.

[5] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, arXiv preprint, arXiv:1412.6980, 2014.

[6] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones, *Character-level language modeling with deeper self-attention*, arXiv preprint, arXiv:1808.04444, 2018.

[7] Byron Knoll, *CMIX version 17*, `http://www.byronknoll.com/cmix.html`.

[8] Byron Knoll, *`lstm-compress`: data compression using LSTM*, `https://github.com/byronknoll/lstm-compress`.

[9] *The .xz file format*, `https://tukaani.org/xz/format.html`.

[10] Matt Mahoney, *Large Text Compression Benchmark*, `http://www.mattmahoney.net/dc/text.html`.

[11] Sennrich, R., Haddow, B., and Birch, A., *Neural machine translation of rare words with subword units*, arXiv preprint, arXiv:1508.07909, 2015.

History

- March 30, 2019: initial version.
- May 4, 2019: added large2 LSTM model. Corrected large1 LSTM model results.