NNCP v2: Lossless Data Compression with Transformer

Fabrice Bellard

Feb 6, 2021

Abstract

NNCP v2 is a lossless data compression program based on the Transformer machine learning model. It achieves state-of-the-art results on the `enwik9` compression benchmark.

1 Introduction

This article is a follow-up of [1] where we presented NNCP, a lossless data compressor based on neural network models. The NNCP Transformer model was improved so that it achieves better compression ratios at the expense of a larger computational cost. The model parameters were chosen so that it is possible to achieve the published results on a personnal computer with a single GPU in a few days.

The source code is available at `https://bellard.org/nncp`.

2 Algorithm description

2.1 Transformer Model

The model is based on the Transformer XL model defined in [2]. The following modifications were made:

- Learned relative positional embeddings are used instead of sinusoid relative positional embeddings. Sinusoid relative positional embeddings are slower and only useful if the memory length at evaluation time is different from the one at training time.

- Untied embeddings give better results.

- The bias v_i of the learned relative positional embeddings is multiplied by a scaling factor $sqrt(d_{model})$ to improve the initial convergence speed. The bias v_i are also identical in all the layers.
- The GELU activation [9] is used instead of ReLU in the feed-forward layer.
- All the weights (except the biases and layer norm weights) are initialized to the same value. But the weights of the second linear transform of the feed-forward layer are scaled by $sqrt(\frac{d_{model}}{d_{inner}})$ to increase the convergence speed.
- No dropout is used in normal operation. Dropout is only employed in the retraining phase.

2.2 Preprocessor

We reused the preprocessor of NNCP v1 [1].

2.3 Training details

Unlike in [3] where only the compression side was considered, we also want to decompress in a reasonnable amount of time. Hence we employ a model which has identical computational steps in the encoder and decoder. It is achieved by encoding or decoding a single symbol (or batch of symbols) per training step. So there is a large overlap between the training segments. In our examples we use training segments of 192 successive symbols. A large batch size (64) is employed in order to exploit more parallelism. The previous NNCP version already used an overlap between the training segments and the idea was further improved in [6]. It is especially interesting with GPUs where the multiplication of large matrices is proportionally much faster than the one of small matrices.

Even with our small improvements to the Transformer XL model, the convergence is still slower than our previous LSTM model. So in order to get a better compression ratio, we *retrain* the model using the already decompressed data at regular intervals. It is equivalent to training the model on several epochs when doing conventional machine learning. This idea was presented in [3]. In order to avoid overfitting, dropout is used in the retraining phase. Different training parameters are used in the retraining phase because we want to maximize the parallelism. Hence we use a smaller batch size but no longer overlap the training segments. In our example, an equivalent of 20 epochs is employed (the past 10 Msymbols are retrained every 500 ksymbols).

We use the Adam optimizer[4] with $\beta_1 = 0$, $\beta_2 = 0.9999$ and $\epsilon = 10^{-9}$ (hence it is equivalent to RMSProp with a bias correction). Using a different Adam context

Program or model	Compr. Size (bytes)	Ratio (bpb)
`gzip -9`	36 445 248	2.92
`xz -9` [7]	24 865 244	1.99
CMIX (v18) [5]	14 838 332	1.19
NNCP v1	16 292 774	1.30
NNCP v2 (base)	15 600 675	1.25
NNCP v2 (large)	15 020 691	1.20

Table 1: Compression results for `enwik8`.

Program or model	Compr. Size (bytes)	Ratio (bpb)	Compr. Speed (kB/s)
`gzip -9`	322 591 995	2.58	17400
`xz -9` [7]	197 331 816	1.58	1020
CMIX (v18) [5]	115 714 367	0.926	1.66
NNCP v1	119 167 224	0.953	1.05
NNCP v2 (base)	114 217 584	0.914	3.25
NNCP v2 (large)	112 219 309	0.898	1.94

Table 2: Compression results for `enwik9`. More complete results can be found in [8].

for the normal and retraining phase is important to get good results because the gradient norms are different.

We also employ gradient normalization which is essential to avoid divergence.

No warm up phase is used. The learning rate is linearly decreased during the training.

The exact parameters are available in the source code.

2.4 Implementation

The algorithm is implemented using PyTorch so that it can easily run on a GPU. We use the deterministic mode of PyTorch to guaranty that the model is identical in the encoding and decoding phases. It is guaranted only if the code is running with the exact same hardware and software versions. 16 bit floating point operations were used to decrease the running time.

3 Results

The results in bytes and bpb (bits per input byte) are given in table 1 and 2 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 [5], 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 [2] because:

- The result is the average bpb over the whole file instead of the test dataset.
- 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` [7], 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.

For NNCP v2, the compression speed was tested with a RTX 3090 GPU. The other programs do not require a GPU. The NNCP v2 decompression speed is similar to its compression speed.

Regarding the compression ratio, we do not reach the performance of CMIX (1.20 versus 1.19 bpb) on `enwik8` but improved the result compared to NNCP v1.

For `enwik9`, NNCP v2 does better than CMIX with a much simpler model. The number of operations is higher but it can still be run in a few days on a personnal computer with a GPU. The `base` model contains 56M parameters. The `large` model contains 187M parameters. More exhaustive hyperparameter tuning is worth investigating.

4 Conclusion

We presented the first practical Transformer implementation able to outperform the best text compression programs on the `enwik9` benchmark [8]. Unlike most today's state-of-the-art natural language processing results, it is achieved on a desktop PC with a single GPU.

References

[1] Fabrice Bellard, *Lossless Data Compression with Neural Networks*, `https://bellard.org/nncp/nncp.pdf`.

[2] 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.

[3] Gautier Izacard, Armand Joulin, Edouard Grave, *Lossless Data Compression with Transformer*, `https://openreview.net/forum?id=Hygi7xStvS`.

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

[5] Byron Knoll, *CMIX version 18*, `http://www.byronknoll.com/cmix.html`.

[6] Byron Knoll, *tensorflow-compress v3*, `https://github.com/byronknoll/tensorflow-compress`.

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

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

[9] Dan Hendrycks, Kevin Gimpel, *Gaussian Error Linear Units (GELUs)*, arXiv preprint, arXiv:1606.08415, 2016.

History

- Jan 3, 2021: initial version.
- Feb 6, 2021: added large model.