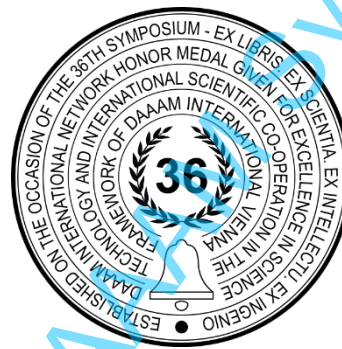


OPTIMISATION OF INFORMATION ENCODING USING MACHINE LEARNING: A COMPARATIVE STUDY OF HUFFMAN CODING AND LSTM NETWORKS

Boris Drasko, Gordana Jugo & Kresimir Rakic



This Publication has to be referred as: Drasko, B[oris]; Jugo, G[ordana] & Rakic, K[resimir] (2026). Optimisation of Information Encoding Using Machine Learning: A Comparative Study of Huffman Coding and LSTM Networks, Proceedings of the 37th DAAAM International Symposium, pp.xxxx-xxxx, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-xx-x, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/37th.daaam.proceedings.xxx

Abstract

This study evaluates how much a context-sensitive Long Short-Term Memory (LSTM) probability model can reduce expected character-level code length relative to static Huffman coding. A reproducible CPU-only prototype was implemented in Python and evaluated on the 1,115,394-character Tiny Shakespeare corpus using a chronological 90%/5%/5% training-validation-test split. A Huffman tree estimated from the training partition provided a zero-order prefix-code baseline. A 419,265-parameter LSTM used 128-character contexts to predict the next character; cross-entropy was converted from nats to model-based bits per character (bpc). On the held-out test partition, Huffman yielded an average prefix-code length of 4.8908 bpc, while the LSTM yielded an estimated ideal code length of 2.3592 bpc, reductions of 51.8% relative to Huffman and 70.5% relative to eight-bit ASCII. Model fitting required 8,491.87 s on a CPU-only laptop. The results demonstrate the value of contextual probability modelling, but do not constitute an end-to-end compressor because arithmetic or range coding and model-transfer overhead were not included.

Keywords: information encoding; lossless data compression; Huffman coding; entropy coding; Long Short-Term Memory; character-level language modelling; neural probability modelling; bits per character.

1. Introduction

Lossless source coding improves storage and transmission efficiency by representing predictable structure with fewer bits while preserving exact reconstruction.

Classical symbol coders, including Huffman and Shannon-Fano coding, assign codewords from marginal symbol frequencies. They are transparent and inexpensive, but a zero-order model cannot exploit the dependence of the next character on its preceding context. Neural sequence models offer a complementary approach by learning conditional probability distributions from data.

The research question is therefore: under one chronological data split and a common set of evaluated target characters, how does the code-length potential of a context-sensitive LSTM model [1] compare with a zero-order Huffman baseline, and what computational and model-size costs accompany the neural approach? The contribution is a reproducible character-level comparison that explicitly separates realised Huffman prefix-code length from the model-based expected code length implied by LSTM probabilities.

2. Theoretical Background

Shannon [2] established the information-theoretic limits of lossless source coding. For a source with known statistics, entropy is the lower bound on average code length under the assumptions of the source-coding theorem. For dependent sequences, the relevant quantity is the entropy rate or an appropriate conditional entropy, rather than only the entropy of isolated symbols.

Huffman [3] formulated an algorithm for constructing an optimal binary symbol-by-symbol prefix code for a given probability distribution. The two least probable symbols are repeatedly merged to form a binary tree. If H_0 denotes zero-order entropy and L is the mean Huffman code length, then $H_0 \leq L < H_0 + 1$ for a non-degenerate source. This optimality is restricted to the chosen class of prefix codes and does not imply optimality for sources with exploitable context.

Shannon-Fano coding also assigns shorter codewords to more probable symbols, but its top-down partitioning procedure does not guarantee a minimum expected prefix-code length. Comparative experiments reported by Viraktamath et al. [4] found Huffman coding at least as efficient as the examined Shannon and Shannon-Fano variants.

Both Huffman and Shannon-Fano codes are prefix-free and therefore uniquely decodable, which supports exact reconstruction in lossless text coding [5].

Experiments on text files and resource-constrained mobile devices likewise show that Huffman coding can achieve lower average code length than Shannon-Fano coding under the tested symbol distributions [6]. These results motivate its use here as a strong, simple zero-order baseline rather than as a claim that Huffman dominates all lossless compressors.

Accordingly, the present experiment uses Huffman coding as the classical symbol-level reference and tests whether conditional neural probabilities can reduce expected code length on the same target characters.

3. Related Work

Prediction-based lossless compressors usually contain two distinct components: a probability model and an entropy coder. The probability model estimates the next-symbol distribution, while arithmetic or range coding converts those probabilities into a reversible bitstream. Recent work further formalises this prediction-compression equivalence and evaluates language models as general-purpose compressors [7]. This distinction is essential when comparing language-model cross-entropy with realised compressed-file size.

Goyal et al. [8] implemented this design in DeepZip by combining recurrent neural predictors with arithmetic coding for synthetic, textual, and genomic sequences. Their results demonstrate that learned probability models can outperform Gzip on several real datasets, while also showing that model storage and decoding symmetry are part of an end-to-end compressor.

Dhulavvagol et al. [9] similarly combined recurrent neural networks with arithmetic coding for lossless text compression. The reported gains support hybrid neural-entropy coding, although results depend on the corpus, predictor architecture, model overhead, and coding implementation.

Bellard [10] developed practical neural lossless compressors based on LSTM and Transformer models. His LSTM variants use online, deterministic probability prediction with arithmetic coding and achieved 1.34 bits per input byte on the enwik8 benchmark for the largest reported LSTM configuration. These results are not directly comparable with held-out language-model bpc because the training protocol, symbol representation, model adaptation, preprocessing, and accounting rules differ.

Building on this literature, the present study isolates the probability-modelling question in a lightweight CPU experiment. It compares a realised Huffman prefix-code length with the ideal code length implied by held-out LSTM probabilities, while explicitly refraining from reporting the latter as an actual compressed-file size.

4. Software Prototype and Development

4.1. Software Architecture and Workflow

A software prototype was developed to compare a zero-order Huffman baseline with a context-sensitive LSTM probability model. The Huffman branch returns a realised average prefix-code length. The LSTM branch returns cross-entropy, which is converted into an estimated ideal code length under a hypothetical entropy coder.

The two branches used the same chronological partitions and exactly the same target characters after the first 128 context symbols of each partition. Runtime was reported as descriptive computational cost rather than as a complete speed comparison between the methods. The common workflow is summarised in Figure 1.

4.2. Dataset

The Tiny Shakespeare corpus distributed with Karpathy's character-level recurrent neural network repository [11] was used for training and evaluation. The downloaded text contained 1,115,394 ASCII characters drawn from several Shakespeare plays and a vocabulary of 65 unique characters.

The corpus was split chronologically into 1,003,854 training characters (90%), 55,769 validation characters (5%), and 55,771 test characters (5%). The training partition fitted the model and constructed the Huffman codebook; validation monitored training and selected the checkpoint; test results were inspected only after model selection.

No characters were shuffled across partition boundaries. Within the training generator, sample order was shuffled between epochs, but every input used only its preceding 128 characters from the same partition.

4.3. Vocabulary Construction

The deterministic vocabulary was constructed from the training partition only. Character-to-index (stoi) and index-to-character (itos) mappings provided reversible conversion, and all validation and test characters were confirmed to be covered by the 65-character training vocabulary.

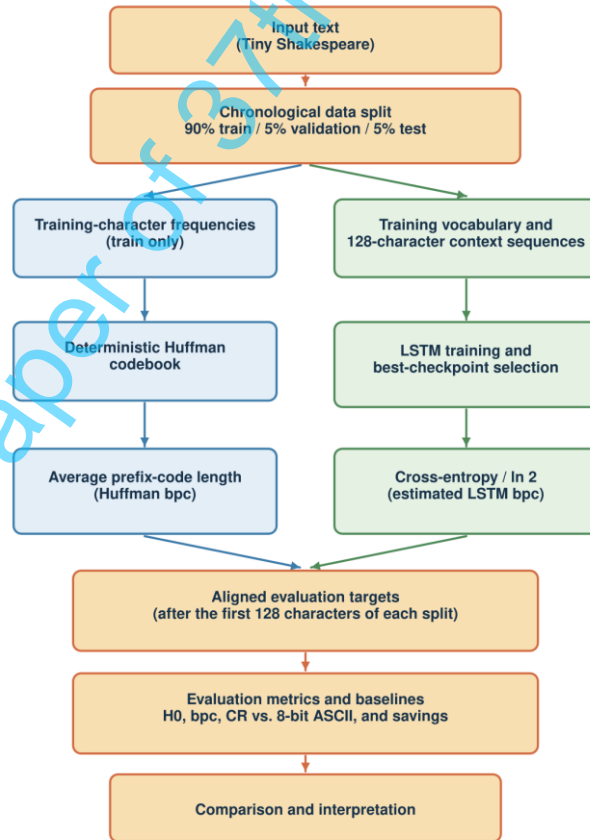


Fig. 1. Experimental workflow for the Huffman and LSTM comparison

Figure 1 presents the experimental evaluation workflow. Both branches use the same chronological corpus split and aligned target positions. Their outputs are compared in bits per character (bpc) and relative to an eight-bit ASCII reference. In the LSTM branch, cross-entropy is converted into an estimated, model-based code length; no compressed bitstream is produced.

4.4. Huffman Baseline

Character frequencies from the training partition were used to construct one deterministic binary Huffman tree. The resulting code lengths were then held fixed when evaluating training, validation, and test targets, preventing information from the held-out partitions from entering the codebook.

For parity with the LSTM, metrics were computed after the first 128 characters of each partition, yielding 1,003,726 training, 55,641 validation, and 55,643 test targets. Reported measures were empirical zero-order entropy (H_0), mean Huffman code length, bpc, and compression ratio relative to eight-bit ASCII. Huffman tree and header overhead were excluded.

4.5. LSTM Model

The LSTM probability model was implemented in TensorFlow/Keras. Each sample contained 128 consecutive character indices and one next-character target, with stride 1. This sequence-to-one construction counts every eligible target character exactly once during evaluation.

The architecture of the model:

- An embedding layer with 128-dimensional vectors;
- One LSTM layer with 256 hidden units (`return_sequences=False`);
- A dense softmax layer with 65 units to produce next-character probabilities.

The network had 419,265 trainable parameters (1.60 MiB as raw float32 values) trained with batch size 256, Adam with initial learning rate of 0.001 and sparse categorical cross-entropy. We used deterministic TensorFlow operations and a fixed random seed of 42. Training was done over three epochs. The run was monitored by EarlyStopping (patience 2), ReduceLROnPlateau (patience 1), ModelCheckpoint and CSV logging. The validation loss improved after every epoch, so training stopped on the preset maximum and epoch 3 was preserved.

4.6. Evaluation Metrics and Protocol

Both methods were evaluated in bits per character (bpc). Huffman bpc is the realised mean length of fixed prefix codewords. LSTM bpc is the model-based ideal code length obtained by dividing sparse categorical cross-entropy in nats per target character by $\ln(2)$. It approximates the length attainable by an ideal entropy coder supplied with the same probabilities.

For either method, the reported compression ratio is $\text{bpc}/8$ and savings relative to ASCII are $100 \times (1 - \text{bpc}/8)$. The LSTM reduction relative to Huffman is $100 \times (1 - \text{bpc_LSTM}/\text{bpc_Huffman})$. These ratios compare code-length estimates and do not include byte packing, file headers, the Huffman codebook, or the LSTM model.

Model-fitting and fixed-checkpoint evaluation times were recorded for the LSTM. Huffman runtime was not used for an end-to-end speed comparison, because the principal experiment concerns code length and the two workflows have different training and coding stages.

4.7. System Configuration

Experiments ran on a Windows 11 laptop with an AMD Ryzen 7 5700U CPU, 16 GB RAM, and integrated AMD Radeon graphics. TensorFlow detected no GPU, so model fitting and evaluation used the CPU.

The software environment comprised Python 3.11.9, NumPy 2.0.2, TensorFlow 2.18.0, and Keras 3.7.0 in a Jupyter Notebook within Visual Studio Code. Deterministic TensorFlow operations were enabled.

5. Results

The results are presented for the two evaluated approaches. The first part analyses Huffman coding as a traditional entropy-based method, while the second examines the LSTM model as a learned probability model.

5.1. Huffman Coding Results

Table 1 presents the results obtained using Huffman coding on the Tiny Shakespeare dataset. Relative savings were calculated using ASCII encoding, which uses 8 bits per character, as a reference point.

Partition	Empirical H0 (bpc)	Huffman (bpc)	Compression ratio
Train	4.7740	4.8114	0.6014
Validation	4.7914	4.8444	0.6056
Test	4.8298	4.8908	0.6114

Table 1. Huffman coding results on the Tiny Shakespeare dataset

Across the three partitions, Huffman required from 4.8114 to 4.8908 bpc, corresponding to compression ratios from 0.6014 to 0.6114 relative to ASCII. The mean code length was from 0.0374 to 0.0610 bpc above empirical H0, as expected from integer-length prefix coding and modest distribution differences between the training-derived codebook and held-out text.

These values are strong for a 65-symbol zero-order baseline, but the fixed codeword for each character cannot change with preceding context.

5.2. LSTM Model Results

The LSTM was trained for the prescribed maximum of three epochs. Validation loss decreased at every epoch, from 1.7009 to 1.5203 nats per target character, and epoch 3 produced the lowest observed validation loss. Consequently, neither early stopping nor learning-rate reduction was triggered. Figure 2 shows the recorded training history.

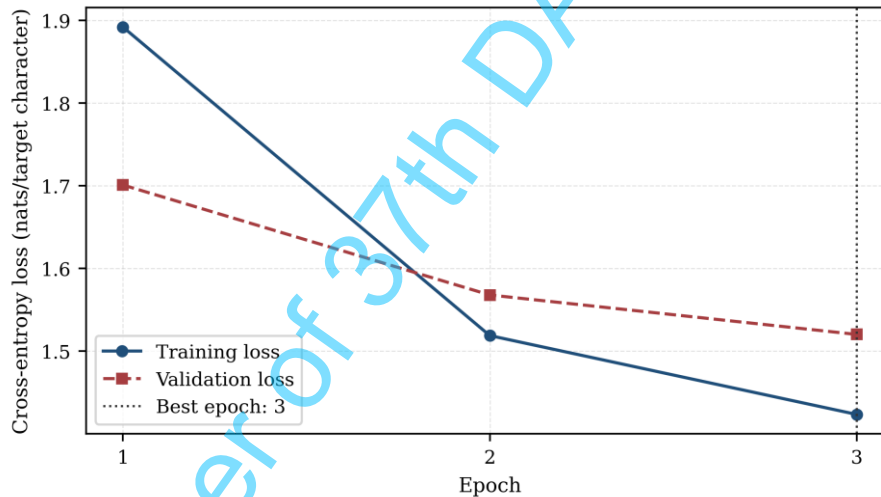


Fig. 2. Training and validation cross-entropy across the three epochs

The training curve reports metrics accumulated while parameters changed within each epoch. Table 2 instead reports a separate evaluation of the fixed epoch-3 checkpoint, so its training loss is not expected to equal the last point of the training curve. The continued validation improvement shows no observed validation deterioration within the three-epoch run, but does not establish full convergence.

Partition	Targets	Loss (nats/char)	Estimated bpc	Estimated CR	Reduction vs. Huffman
Train	1,003,726	1.373014	1.9808	0.2476	58.83%
Validation	55,641	1.520269	2.1933	0.2742	54.73%
Test	55,643	1.635293	2.3592	0.2949	51.76%

Table 2. LSTM model results and estimated compression ratios

Using the same epoch-3 checkpoint for every partition, estimated LSTM code length increased from 1.9808 bpc on training targets to 2.1933 bpc on validation and 2.3592 bpc on test. On the test partition, this represents a 51.76% reduction

relative to Huffman and a 70.51% reduction relative to eight-bit ASCII. Model fitting required 8,491.87 s; fixed-checkpoint evaluation required 1,565.58 s, 56.28 s, and 56.92 s for train, validation, and test, respectively.

The selected model contained 419,265 parameters. Its raw float32 parameters occupied 1.60 MiB, while the saved Keras checkpoint occupied 4.82 MiB. These sizes are excluded from Table 2 and are considered explicitly in the limitations.

6. Discussion

6.1. Comparative Analysis

The results expose a clear trade-off: the context-sensitive model assigns much shorter ideal code lengths to Tiny Shakespeare targets, whereas Huffman provides a complete and inexpensive prefix code without a learned model.

Relative to the eight-bit ASCII reference, Huffman ratios ranged from 0.6014 to 0.6114. The LSTM estimates ranged from 0.2476 to 0.2949. On validation and test targets, the estimated LSTM reductions relative to Huffman were 54.73% and 51.76%, respectively.

Huffman coding is appropriate when simplicity, deterministic decoding, and low computational cost are priorities. It is optimal within binary symbol-by-symbol prefix codes for the supplied marginal distribution, but it cannot use the preceding 128-character context represented by the LSTM.

The LSTM reduces cross-entropy by conditioning each prediction on previous characters, at the cost of 419,265 learned parameters and substantial CPU time. This benefit is therefore a probability-modelling result, not yet a complete compression result.

The widening train-to-test gap (1.9808 versus 2.3592 estimated bpc) indicates a generalisation penalty under the chronological split, but the held-out estimate remains substantially below the zero-order Huffman baseline.

Figure 3 summarises the comparison while preserving the distinction between realised Huffman lengths and model-based LSTM estimates.

A fitting time of 8,491.87 s on the CPU illustrates the main computational cost. Because the study did not implement encoding and decoding for the LSTM probabilities, and did not report a matched end-to-end Huffman runtime, no speed superiority is claimed for either method.

Figure 3 compares ASCII, Huffman, and LSTM code lengths on identical target sets. Lower bars indicate shorter code length; the LSTM bars remain estimates under ideal entropy coding.

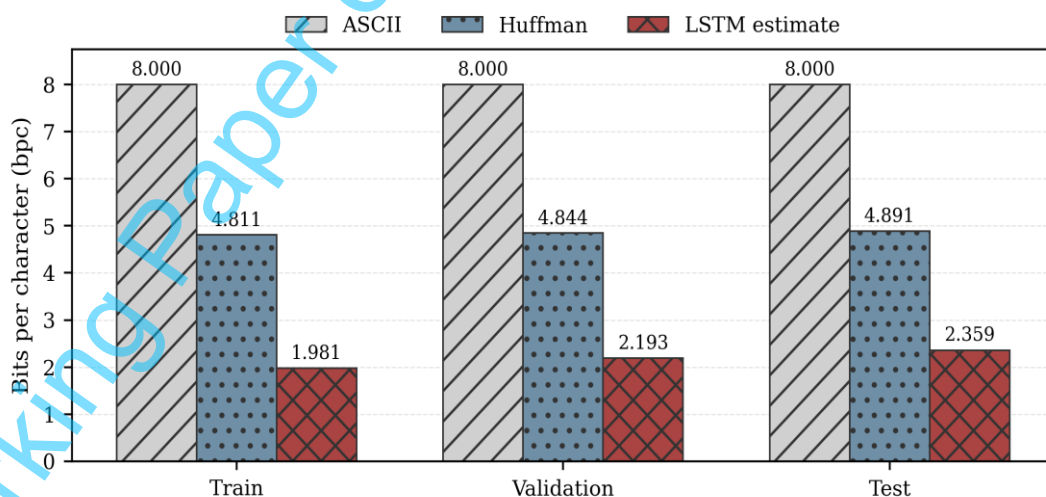


Fig. 3. Average code-length comparison; LSTM values are model-based estimates

Under this controlled setup, contextual probability modelling produced substantially lower held-out cross-entropy than a fixed-symbol Huffman code. The evidence supports the LSTM as a stronger probability model for this corpus, but it does not demonstrate a smaller standalone compressed file until coder and model overhead are included.

6.2. Information-Theoretic Interpretation

The empirical H_0 values, from 4.7740 to 4.8298 bpc, describe isolated-character uncertainty. They are not lower bounds for a model that conditions on context: for a dependent source, conditional entropy and entropy rate can be substantially smaller than H_0 .

The Huffman results lie close to H_0 because the code uses one fixed integer-length codeword per character. By contrast, the LSTM supplies a different probability distribution for each target position based on its 128-character history.

For each target character, the contribution to ideal code length is the negative base-2 logarithm of the conditional probability assigned to that character from its preceding 128-character context. Averaging this quantity over targets yields the reported model-based bpc.

The LSTM-based results reported in this study represent estimated code lengths derived from the predicted symbol probabilities and therefore do not correspond to an actual compressed bitstream. To obtain a practical lossless compressor, these probabilities would have to be coupled with an entropy coder [8], [9]. One possible solution is non-stationary arithmetic coding, in which the coding intervals are continuously adapted according to the probability distribution produced by the neural model for each successive character. Since the LSTM distribution changes after every prediction, the arithmetic coder would need to update its cumulative frequency representation accordingly. In practice, this introduces additional computational and implementation overhead due to probability quantization, frequency normalization, finite-precision arithmetic, coder renormalization, and synchronization between the neural predictor and the entropy-coding stage.

Asymmetric Numeral Systems (ANS) provides another suitable approach for converting the predicted probabilities into a real compressed bitstream. ANS can achieve compression efficiency comparable to arithmetic coding while using a single integer state and relatively simple state updates [12]. In the present non-stationary setting, however, the continuously changing probability distribution produced by the LSTM must first be converted into discrete frequency counts suitable for the entropy coder. A practical implementation could therefore use the range variant of Asymmetric Numeral Systems (rANS) with quantized integer frequencies [13]. If coding tables or frequency representations were reconstructed for every predicted character, this would introduce additional computational overhead; alternatively, probability distributions could be reused or updated in blocks, at the cost of some coding efficiency. Further overhead arises from probability quantization, frequency normalization, state renormalization, and synchronization between the neural predictor and the entropy coder. Consequently, the size of a real compressed bitstream produced by either arithmetic coding or ANS would be expected to be slightly larger than the ideal code length estimated directly from the LSTM probabilities. Implementing and benchmarking such an end-to-end entropy-coding pipeline is therefore a natural direction for future work.

Consequently, the valid conclusion is narrower than claiming universal compressor superiority: the trained LSTM assigned lower expected conditional code length than the training-derived zero-order Huffman code on this specific chronological split.

6.3. Implications and Limitations

The main practical implication is that context modelling can provide a large code-length advantage for structured text when a compatible model is already available to both encoder and decoder.

The result motivates integrating lightweight neural predictors with entropy coders for sequential data. It does not, by itself, establish equivalent gains for genomic, audio, or real-time communication data, which require separate experiments.

The neural approach also introduces material resource costs. A recent DAAAM study likewise notes that neural-network methods can require substantial data, memory, and processing capacity [14]. In the present experiment, fitting alone required more than two hours on the CPU-only laptop.

Data availability is another practical constraint. Recent DAAAM research demonstrates that useful machine-learning models can be developed from small datasets, although broader datasets remain important for improving generalisation [15]. Hybrid solutions that combine machine learning with established engineering methods have also been proposed to improve performance while maintaining computational efficiency [16].

Several limitations constrain interpretation. First, no arithmetic or range coder was implemented, so LSTM bpc is an ideal model-based estimate rather than a measured compressed-file size.

Second, neither the Huffman codebook nor LSTM model was charged to the basic ratios. If the raw float32 parameters were transmitted only with the 55,643 test targets, they would add 241.12 bpc; the 4.82-MiB Keras checkpoint would add 727.35 bpc. The reported neural advantage therefore assumes a pre-shared or amortised model.

Third, the study used one corpus, one chronological split, one deterministic seed, three epochs, and no systematic hyperparameter search or repeated-run uncertainty analysis. Because the best validation value occurred at the final allowed epoch, full convergence cannot be claimed.

Future work should implement deterministic arithmetic or range encoding and decoding, include all headers and model-transfer costs, and compare end-to-end size and runtime with Huffman, Gzip, xz, and context-based compressors on multiple corpora.

Further experiments should also evaluate additional seeds, longer but hardware-feasible training schedules, smaller or quantised models, and scenarios in which the trained model is reused across enough data to amortise its storage cost.

7. Conclusion

This study established a reproducible CPU-based comparison between a training-derived Huffman prefix code and a character-level LSTM probability model on the Tiny Shakespeare corpus.

On the held-out test targets, Huffman yielded 4.8908 bpc (CR = 0.6114), whereas the fixed epoch-3 LSTM checkpoint yielded an estimated ideal code length of 2.3592 bpc (CR = 0.2949). The model-based reductions were 51.76% relative to Huffman and 70.51% relative to eight-bit ASCII.

The result answers the research question for the stated configuration: incorporating a 128-character context greatly reduced held-out expected code length, but required 419,265 parameters and 8,491.87 s of CPU fitting time.

The central conclusion is therefore about probability modelling rather than a finished compressor. Better conditional prediction can lower ideal code length, but practical lossless compression must also account for deterministic coding, model availability, headers, precision, size, and decoding cost.

The next step is to couple the predictor to an arithmetic or range coder, verify byte-for-byte reconstruction, and measure total compressed size including model and metadata overhead. Evaluation on several corpora and against stronger end-to-end baselines will determine whether the observed advantage generalises and can be made practical.

8. References

- [1] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory, *Neural Computation*, Vol. 9, No. 8, pp. 1735-1780, ISSN 0899-7667, DOI: 10.1162/neco.1997.9.8.1735
- [2] Shannon, C. E. (1948). A mathematical theory of communication, *Bell System Technical Journal*, Vol. 27, No. 3, pp. 379-423, and Vol. 27, No. 4, pp. 623-656, ISSN 0005-8580, DOI: 10.1002/j.1538-7305.1948.tb01338.x and 10.1002/j.1538-7305.1948.tb00917.x
- [3] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes, *Proceedings of the IRE*, Vol. 40, No. 9, pp. 1098-1101, ISSN 0096-8390, DOI: 10.1109/JRPROC.1952.273898
- [4] Viraktamath, S. V.; Attimarad, G. V. & Bhate, G. (2011). Impact of selection of source coding technique on the efficiency, *Proceedings of the 2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies*, pp. 638-641, IEEE, DOI: 10.1109/ICSCCN.2011.6024629
- [5] Jayasankar, U.; Thirumal, V. & Ponnuram, D. (2021). A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications, *Journal of King Saud University - Computer and Information Sciences*, Vol. 33, No. 2, pp. 119-140, ISSN 1319-1578, DOI: 10.1016/j.jksuci.2018.05.006
- [6] Mantoro, T.; Ayu, M. A. & Anggraini, Y. (2017). The performance of text file compression using Shannon-Fano and Huffman on small mobile devices, *Proceedings of the 2017 International Conference on Computing, Engineering, and Design*, pp. 1-5, IEEE, DOI: 10.1109/CED.2017.8308127
- [7] Deletang, G.; Ruoss, A.; Duquenne, P.-A.; Catt, E.; Genewein, T.; Mattern, C.; Grau-Moya, J.; Wenliang, L. K.; Aitchison, M.; Orseau, L.; Hutter, M. & Veness, J. (2024). Language modeling is compression, *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*, Available from: https://proceedings.iclr.cc/paper_files/paper/2024/hash/3cbf627fa24fb6cb576e04e689b9428b-Abstract-Conference.html Accessed: 2026-09-19
- [8] Goyal, M.; Tatwawadi, K.; Chandak, S. & Ochoa, I. (2018). DeepZip: Lossless data compression using recurrent neural networks, Available from: <https://arxiv.org/abs/1811.08162> Accessed: 2026-09-18, DOI: 10.48550/arXiv.1811.08162
- [9] Dhulavagol, P. M.; Gadagkar, A.; KJ, A.; Hegade, G.; Poonia, R. & Totad, S. G. (2024). Lossless text compression using recurrent neural networks, *Procedia Computer Science*, Vol. 235, pp. 3340-3349, ISSN 1877-0509, DOI: 10.1016/j.procs.2024.04.315
- [10] Bellard, F. (2019). Lossless data compression with neural networks, Available from: <https://bellard.org/nncp/nncp.pdf> Accessed: 2026-09-08

- [11] Karpathy, A. (2015). char-rnn: Multi-layer Recurrent Neural Networks (LSTM, GRU, RNN) for character-level language models in Torch, Available from: <https://github.com/karpathy/char-rnn> Accessed: 2026-09-08
- [12] Duda, J. (2013). Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding, Available from: <https://arxiv.org/abs/1311.2540> Accessed: 2026-09-18, DOI: 10.48550/arXiv.1311.2540
- [13] Townsend, J. (2020). A tutorial on the range variant of asymmetric numeral systems, Available from: <https://arxiv.org/abs/2001.09186> Accessed: 2026-09-18, DOI: 10.48550/arXiv.2001.09186
- [14] Josipovic, E[ma]; Regvart, D[amir]; Kapulica, A[na] & Kopal, R[obert] (2024). Machine Learning for Threat Detection in SOCs, Proceedings of the 35th DAAAM International Symposium, pp.0279-0285, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-44-0, ISSN 1726-9679, Vienna, Austria, DOI: 10.2507/35th.daaam.proceedings.038
- [15] Zaman, A[[] G[M]; Mahmoud, N[esma]; Virro, I[ndrek]; Liivapuu, O[lga]; Lillerand, T[ormi]; Roy, K[allol] & Olt, J[uri] (2024). Learning with Small Data: A Novel Framework for Blueberry Root Collar Detection, Proceedings of the 35th DAAAM International Symposium, pp.0312-0321, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-44-0, ISSN 1726-9679, Vienna, Austria, DOI: 10.2507/35th.daaam.proceedings.043
- [16] Bahrami, A[mirhossein] & Potekhin, V[yacheslav] (2025). Hybrid Machine Learning-Enhanced PID Controller Optimization for Industrial Process Control, Proceedings of the 36th DAAAM International Symposium, pp.0067-0073, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-47-1, ISSN 1726-9679, Vienna, Austria, DOI: 10.2507/36th.daaam.proceedings.009