SEQUENTIAL NEURAL TEXT COMPRESSION

Preprint. Final version in IEEE Transactions on Neural Networks, 7(1):142-146, 1996

Jürgen Schmidhuber* Stefan Heil

Institut für Informatik, H2
Technische Universität München
80290 München, Germany

May 19, 1994

Abstract

The purpose of this paper is to show that neural networks are promising tools for data compression without loss of information. We combine predictive neural nets and statistical coding techniques to compress text files. We apply our methods to short newspaper articles and obtain compression ratios exceeding those of widely used Lempel-Ziv algorithms (which build the basis of the UNIX functions "compress" and "gzip"). The main disadvantage of our methods is that they are three orders of magnitude slower than standard methods.

I. INTRODUCTION

Text compression is important (e.g. [1]). It is cheaper to communicate compressed text files instead of original text files. Moreover, compressed files are cheaper to store.

For such reasons, various text encoding algorithms have been developed, plus the corresponding decoding algorithms. A text encoding algorithm takes a text file and generates a shorter compressed file from it. The compressed file contains all the information necessary to restore the original file, which can be done by calling the corresponding decoding algorithm (unlike with image compression, text compression typically requires *loss-free* compression). The most widely used text compression algorithms are based on Lempel-Ziv techniques, e.g. [13]. Lempel-Ziv compresses symbol strings sequentially, essentially replacing substrings by pointers to equal substrings encountered earlier. As the file size goes to infinity, Lempel-Ziv becomes asymptotically optimal in a certain information theoretic sense [12].

The average ratio between the lengths of original and compressed files is called the average compression ratio. We cite a statement from Held's book [2], where he refers to text represented by 8 bits per character:

> "In general, good algorithms can be expected to achieve an average compression ratio of 1.5, while excellent algorithms based upon sophisticated processing techniques will achieve an average compression ratio exceeding 2.0."

This paper will show that neural networks can be used to design "excellent" text compression algorithms.

Outline of paper. Section II describes the basic approach combining neural nets and the technique of predictive coding. Section III focuses on the details of a neural predictor of conditional probabilities. In addition, section III describes three alternative coding techniques to be used in conjunction with the predictor. Section IV presents comparative simulations. Section V discusses limitations and extensions.

*Email: schmidhu@informatik.tu-muenchen.de

II. BASIC APPROACH

We combine neural nets, standard statistical compression methods like Huffman-Coding (e.g. [2]) and Arithmetic Coding (e.g. [11]), and variants of the "principle of history compression" [7] [5]. The main ideas of the various alternatives will be explained in section III.

All our methods are instances of a strategy known as "predictive coding" or "model-based coding". We use a neural predictor network P which is trained to approximate the conditional probability distribution of possible characters, given previous characters. P's outputs are fed into coding algorithms that generate short codes for characters with low information content (characters with high predicted probability) and long codes for characters conveying a lot of information (highly unpredictable characters).

Why not use a look-up table instead of a network? Because look-up tables tend to be extremely inefficient. A look-up table requires k^{n+1} entries for all the conditional probabilities of k possible characters, given n previous characters. In addition, a special procedure is required for dealing with previously unseen combinations of input characters. In contrast, the size of a neural net typically grows in proportion to n^2 (assuming the number of hidden units grows in proportion to the number of inputs), and its inherent "generalization capability" is going to take care of previously unseen combinations of input characters (hopefully by coming up with good predicted probabilities).

We will make the distinction between *on-line* and *off-line* variants of our approach. With *off-line* methods, P is trained on a separate set F of training files. After training, the weights are frozen and copies of P are installed at all machines functioning as message receivers or senders. From then on, P is used to encode and decode unknown files without being changed any more. The weights become part of the code of the compression algorithm. The storage occupied by the network weights does not have to be taken into account to measure the performance on *unknown* files – just like the code for a conventional data compression algorithm does not have to be taken into account.

The *on-line* variants are based on the insight that even if the predictor learns *during* compression, the modified weights need not be sent from the sender to the receiver across the communication channel – as long as the predictor employed for decoding uses exactly the same initial conditions and learning algorithm as the predictor used for encoding (this observation goes back to Shannon). Since on-line methods can adapt to the statistical properties of specific files, they promise significantly better performance than off-line methods. But there is a price to pay: on-line methods tend to be computationally more expensive.

Section IV will show that even *off-line* methods can achieve excellent results. We will briefly come back to on-line methods in the final section of this paper.

III. OFF-LINE METHODS

In what follows, we will first describe the training phase of the predictor network P (a strictly layered feed-forward net trained by back-propagation [10][4]). The training phase is based on a set F of training files. Then we will describe three working off-line variants of "compress" and "uncompress" functions based on P. All methods are *guaranteed* to encode and decode arbitrary unknown text files without loss of information.

A. THE PREDICTOR NETWORK P

Assume that the alphabet contains k possible characters $z_1, z_2, \ldots, z_k$. The (local) representation of z_i is a binary k-dimensional vector $r(z_i)$ with exactly one non-zero component (at the i-th position). P has nk input units and k output units. n is called the "time-window" size. We insert n default characters z_0 at the beginning of each file. The representation of the default character, $r(z_0)$, is the k-dimensional zero-vector. The m-th character of file f (starting from the first default character) is called c^f_m.

For all $f \in F$ and all possible $m > n$, P receives as an input

$$r(c^f_{m-n}) \circ r(c^f_{m-n+1}) \circ \ldots \circ r(c^f_{m-1}),$$

where $\circ$ is the concatenation operator for vectors. P produces as an output P^f_m, a k-dimensional output vector. Using back-propagation [10][4], P is trained to minimize

$$\frac{1}{2} \sum_{f \in F} \sum_{m>n} \parallel r(c^f_m) - P^f_m \parallel^2 . \tag{1}$$

(1) is minimal if P^f_m always equals

$$E(r(c^f_m) \mid c^f_{m-n}, \ldots, c^f_{m-1}), \tag{2}$$

the conditional expectation of $r(c^f_m)$, given $r(c^f_{m-n}) \circ r(c^f_{m-n+1}) \circ \ldots \circ r(c^f_{m-1})$. Due to the local character representation, this is equivalent to $(P^f_m)_i$ being equal to the conditional probability

$$Pr(c^f_m = z_i \mid c^f_{m-n}, \ldots, c^f_{m-1}) \tag{3}$$

for all f and for all appropriate $m > n$, where $(P^f_m)_j$ denotes the j-th component of the vector P^f_m.

In practical applications, the $(P^f_m)_i$ will not always sum up to 1. To obtain outputs satisfying the properties of a proper probability distribution, we normalize by defining

$$P^f_m(i) = \frac{(P^f_m)_i}{\sum_{j=1}^k (P^f_m)_j}. \tag{4}$$

B. METHOD 1

With the help of a copy of P, an unknown file f can be compressed as follows: Again, n default characters are inserted at the beginning. For each character c^f_m $(m > n)$, the predictor emits its output P^f_m based on the n previous characters. There will be a k such that $c^f_m = z_k$. The estimate of $Pr(c^f_m = z_k \mid c^f_{m-n}, \ldots, c^f_{m-1})$ is given by $P^f_m(k)$. The code of c^f_m, the bitstring $code(c^f_m)$, is generated by feeding $P^f_m(k)$ into the Huffman Coding algorithm (see below). $code(c^f_m)$ is written into the compressed file.

Huffman Coding

With a given probability distribution on a set of possible characters, Huffman Coding (e.g. [2]) encodes characters by bitstrings as follows.

Characters correspond to terminal nodes of a binary tree to be built in an incremental fashion. The probability of a terminal node is defined as the probability of the corresponding character. The probability of a non-terminal node is defined as the sum of the probabilities of its sons. Starting from the terminal nodes, a binary tree is built as follows:

Repeat as long as possible:

Among those nodes that are not children of any non-terminal nodes created earlier, pick two with lowest associated probabilities. Make them the two sons of a newly generated non-terminal node.

The branch to the "left" son of each non-terminal node is labeled by a 0. The branch to its "right" son is labeled by a 1. The code of a character c, $code(c)$, is the bitstring obtained by following the path from the root to the corresponding terminal node. Obviously, if $c \neq d$, then $code(c)$ cannot be the prefix of $code(d)$. This makes the code uniquely decipherable. Note that characters with high associated probability are encoded by short bitstrings. Characters with low associated probability are encoded by long bitstrings.

The probability distribution on the characters is not required to remain fixed. This allows for using "time-varying" conditional probability distributions as generated by the neural predictor.

How to decode

The information in the compressed file is sufficient to reconstruct the original file. This is done with the "uncompress" algorithm, which works as follows: Again, for each character c^f_m $(m > n)$, the predictor (sequentially) emits its output P^f_m based on the n previous characters, where the c^f_l with $n < l < m$ were gained sequentially by feeding the approximations $P^f_l(k)$ of the probabilities $Pr(c^f_l = z_k \mid c^f_{l-n}, \ldots, c^f_{l-1})$ into the inverse Huffman Coding procedure. The latter is able to correctly decode c^f_l from $code(c^f_l)$. Note that to correctly decode some character, we first need to decode all previous characters.

C. METHOD 2

Like METHOD 1, but with Arithmetic Coding (see below) replacing the non-optimal Huffman-Coding (a comparison of alternative coding schemes will be given in subsection III E).

Arithmetic Coding

The basic idea of Arithmetic Coding is: a message is encoded by an interval of real numbers from the unit interval $[0, 1[$. The output of Arithmetic Coding is a binary representation of the boundaries of the corresponding interval. This binary representation is incrementally generated during message processing. Starting with the unit interval, for each observed character the interval is made smaller, essentially in proportion to the probability of the character. A message with low information content (and high corresponding probability) is encoded by a comparatively large interval, whose precise boundaries can be specified with comparatively few bits. A message with a lot of information content (and low corresponding probability) is encoded by a comparatively small interval, whose boundaries require comparatively many bits to be specified.

Although the basic idea is elegant and simple, additional technical considerations are necessary to make Arithmetic Coding practicable. See [11] for details.

D. METHOD 3

This section presents another alternative way of "predicting away" redundant information in sequences. Again, we pre-process input sequences by a network that tries to predict the next input, given previous inputs. The input vector corresponding to time step t of sequence p is denoted by $x^p(t)$. The networks real-valued output vector is denoted by $y^p(t)$. Among the possible input vectors, there is one with minimal Euclidean distance to $y^p(t)$. This one is denoted by $z^p(t)$. $z^p(t)$ is interpreted as the deterministic vector-valued prediction of $x^p(t+1)$.

It is important to observe that all information about the input vector $x^p(t_k)$ (at time t_k) is conveyed by the following data: the time t_k, a description of the predictor and its initial state, and the set

$$\{(t_s, x^p(t_s)) \;\; with \;\; 0 < t_s \leq t_k, z^p(t_s - 1) \neq x^p(t_s)\}.$$

In what follows, this observation will be used to compress text files.

Application to Text Compression

Like with METHODs 1 and 2, the "time-window" corresponding to the predictor input is sequentially shifted across the unknown text file. The P^f_m, however, are used in a different way. The character z_i whose representation $r(z_i)$ has minimal Euclidean distance to P^f_m is taken as the predictor's deterministic prediction (if there is more than one character with minimal distance to the output, then we take the one with lowest ASCII value). If c^f_m does not match the prediction, then it is stored in a second file, together with a number indicating how many characters were processed since the *last* non-matching character [7][5]. Expected characters are simply ignored – they represent redundant information. To avoid confusions between unexpected numbers from the original file and numbers indicating how many correct predictions went by since the last wrong prediction, we introduce an escape character to mark unexpected number characters in the second file. The escape character is used to mark unexpected escape characters, too. Finally we apply Huffman-Coding (as embodied by the UNIX function *pack*) to the second file and obtain the final compressed file.

The "uncompress" algorithm works as follows: we first unpack the compressed file by inverse Huffman-Coding (as employed by the UNIX function *unpack*). Then, starting from n default characters, the predictor sequentially tries to predict each character of the original file from the n previous characters (deterministic predictions are obtained like with the compression procedure above.) The numbers in the unpacked file contain all information about which predictions are wrong, and the associated characters tell us how to correct wrong predictions: if the unpacked file indicates that the current prediction is correct, it is fed back to the predictor input and becomes part of the basis for the next prediction. If the unpacked file indicates that the current prediction is wrong, the corresponding entry in the unpacked file (the correct character associated with the number indicating how many correct predictions went by since the last unexpected character) replaces the prediction and is fed back to the predictor input where it becomes part of the basis for the next prediction.

E. COMPARISON OF METHODS 1, 2, 3

With a given probability distribution on the characters, Huffman Coding guarantees minimal expected code length, provided all character probabilities are integer powers of $\frac{1}{2}$. In general, however, Arithmetic Coding works slightly better than Huffman Coding. For sufficiently long messages, Arithmetic Coding achieves expected code lengths arbitrarily close to the information-theoretic lower bound. This is true even if the character probabilities are not powers of $\frac{1}{2}$ (see e.g. [11]).

METHOD 3 is of interest if typical files contain long sequences of predictable characters. Among the methods above, it is the only one that explicitly encodes strings of characters (as opposed to single characters). It does not make use of all the information about conditional probabilities, however.

Once the current conditional probability distribution is known, the computational complexity of Huffman Coding is $O(klogk)$. The computational complexity of Arithmetic Coding is $O(k)$. So is the computational complexity of METHOD 3. In practical applications, however, the computational effort required for all three variants is negligible in comparison to the effort required for the predictor updates.

IV. SIMULATIONS

Our current computing environment prohibits extensive experimental evaluations of the three methods above. On an HP 700 workstation, the training phase for the predictor turns out to be quite time consuming, taking days of computation time. Once the predictor is trained, the method still tends to be on the order of 1000 times slower than standard methods. In many data transmission applications, communication is not expensive enough to justify this in absence of specialized hardware (given the current state of workstation technology). This leads us to recommending special neural net hardware for our approach. The software simulations presented in this section, however, will show that "neural" compression techniques can achieve excellent compression ratios.

We applied our *off-line* methods to German newspaper articles. We compared the results to those obtained with standard encoding techniques provided by the operating system UNIX, namely "pack", "compress", and "gzip". The corresponding decoding algorithms are "unpack", "uncompress", and "gunzip", respectively. "pack" is based on Huffman-Coding (e.g. [2]), while "compress" and "gzip" are based on asymptotically "optimal" Lempel-Ziv techniques, e.g. [13]. It should be noted that "pack", "compress", and "gzip" ought to be classified as *on-line* methods – they adapt to the specific text file they see. In contrast, the competing "neural" methods ran *off-line*, due to time limitations. Therefore our comparison was unfair in the sense that it was biased *against* the "neural" methods. See section V, however, for *on-line* "neural" alternatives.

The training set for the predictor was given by a set of 40 articles from the newspaper *Münchner Merkur*, each containing between 10000 and 20000 characters. The alphabet consisted of $k = 80$ possible characters, including upper case and lower case letters, ciphers, interpunction symbols, and special German letters like "ö", "ü", "ä". P had 430 hidden units. A "true" unit with constant activation 1.0 was connected to all hidden and output units. The learning rate was 0.2. The training phase consisted of 25 sweeps through the training set, taking 3 days of computation time on an HP 700 station. Why just 25 sweeps? On a separate test set, numbers of sweeps between 20 and 40 were empirically found to lead to acceptable performance. Note that a single sweep actually provides many different training examples for the predictor.

The test set consisted of 20 newspaper articles (from the same newspaper), each containing between 10000 and 20000 characters. Of course, the test set did not overlap with the training set. Table 1 lists the average compression ratios and the corresponding variances. Our methods achieved "excellent" performance (according to Held's statement quoted in the introduction). Even METHOD 3 led to an "excellent" compression ratio, although it does not make use of all the information about the conditional probabilities. The best performance was obtained with METHOD 2, which clearly outperformed the strongest conventional competitor, the UNIX "gzip" function based on an asymptotically optimal Lempel-Ziv algorithm. Note that variance goes up (but always remains within acceptable limits) as compression performance improves.

——————————— INSERT TABLE 1 HERE ———————————

The hidden units were actually necessary to achieve good performance. A network without hidden units was not able to achieve average compression ratios exceeding 2.0. The precise number of hidden units appeared to be not very important, though. A network with 300 hidden units achieved performance similar to the one of the network above.

How does a neural net trained on articles from *Münchner Merkur* perform on articles from other sources? *Without retraining the neural predictor*, we applied all competing methods to 10 articles from another German newspaper (the *Frankenpost*). The results are given in table 2.

——————————— INSERT TABLE 2 HERE ———————————

The *Frankenpost* articles were harder to compress for all algorithms. But relative performance remained comparable.

Note that we used quite a small time-window ($n = 5$). In general, larger time windows will make more information available to the predictor. In turn, this will improve the prediction quality and increase the compression ratio. Therefore we expect to obtain even better results for $n > 5$ and for recurrent predictor networks (note that recurrent nets are less limited to time window approach – in principle they can emit predictions based on *all* previous characters). Another reason for optimism is given by a performance comparison with three human subjects who had to predict characters (randomly selected from the test files) from n preceding characters. With $n = 5$, the humans were able to predict 52 percent of all characters, while our predictor predicted 49 percent (the character with the highest predicted probability was taken as the prediction). With $n = 10$, humans were able to predict about 59 percent of all characters. With $n = 15$, humans were able to predict about 63 percent of all characters. We expect that P will remain close to human performance for $n > 5$. More training data, however, are required to avoid overfitting.

V. DISCUSSION

Our results show that neural networks are promising tools for loss-free data compression. It was demonstrated that even *off-line* methods based on small time windows can lead to excellent compression ratios. We have hardly begun, however, to exhaust the potential of the basic approach.

A *disadvantage* of the off-line technique is that it is off-line: the predictor does not adapt to the specific text file it sees. Instead it relies on regularities extracted during the training phase, and on its ability to generalize. This tends to make it language specific. English texts or C-code should be compressed with a predictor different from the one used for German texts (unless one takes the effort and trains the predictor on texts from many different sources, of course).

As mentioned in section II, this limitation is not essential. It is straight-forward to construct *on-line* variants of all three methods described in the previous sections. With these on-line variants, the predictor continues to learn *during* compression. A typical on-line variant proceeds like this: both the sender and the receiver start with *exactly the same* initial predictor. Whenever the sender sees a new character, it encodes it using its current predictor. The code is sent to the receiver who decodes it. Both the sender and the receiver use *exactly the same* learning protocol to modify their weights (for instance: after processing every 1000th character, take the last 10000 symbols to retrain the predictor). The modified weights need not be sent from the sender to the receiver and do not have to be taken into account to compute the average compression ratio. Especially with long unknown text files, the on-line variant should make a big difference. Initial experiments with on-line variants of METHODs 2 and 3 led to additional significant improvements of the compression ratio.

The main disadvantage of both *on-line* and *off-line* variants, however, is their computational complexity. Our current *off-line* implementations are clearly slower than conventional standard techniques, by a factor of about 1000 (but we did not attempt to optimize our systems with respect to speed). And the complexity of an *on-line* method is typically even worse than the one of the corresponding off-line method (the precise slow-down factor depends on the nature of the learning protocol, of course). For this reason, especially the promising on-line variants can be recommended only if special neural net hardware is available. Note, however, that there are *many* commercial data compression applications which rely on specialized electronic chips.

There are a few obvious directions for *future experimental research:* (1) Use larger time windows or recurrent nets – they seem to be very promising even for off-line methods (see the last paragraph of section IV). (2) Thoroughly test the potential of on-line methods. Both (1) and (2) should greatly benefit from fast hardware.

Finally we mention that there are additional interesting applications of neural predictors of conditional probabilities. See [8] for a method that uses a predictor of conditional probabilities to modulate the sequence processing strategy of a separate recurrent network R. This can greatly improve R's ability to detect correlations between events separated by long time lags. See [6] for a method that uses predictors of conditional probabilities to develop factorial codes of environmental input patterns – codes with the property that the code components are statistically independent (see [9] and [3] for applications). This can be useful in conjunction with statistical classifiers that assume statistical independence of their input variables.

VI. ACKNOWLEGMENTS

Thanks to David MacKay for directing our attention towards Arithmetic Coding. Thanks to Gerhard Weiss for useful comments. This research was supported in part by a DFG fellowship to J. Schmidhuber.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] G. Held. *Data Compression.* Wiley and Sons LTD, New York, 1991.

[3] S. Lindstädt. Comparison of two unsupervised neural network models for redundancy reduction. In M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman, and A. S. Weigend, editors, *Proc. of the 1993 Connectionist Models Summer School*, pages 308–315. Hillsdale, NJ: Erlbaum Associates, 1993.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

[5] J. H. Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

[6] J. H. Schmidhuber. Learning factorial codes by predictability minimization. *Neural Computation*, 4(6):863–879, 1992.

[7] J. H. Schmidhuber. Learning unambiguous reduced sequence descriptions. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 291–298. San Mateo, CA: Morgan Kaufmann, 1992.

[8] J. H. Schmidhuber, M. C. Mozer, and D. Prelinger. Continuous history compression. In H. Hüning, S. Neuhauser, M. Raus, and W. Ritschel, editors, *Proc. of Intl. Workshop on Neural Networks, RWTH Aachen*, pages 87–95. Augustinus, 1993.

[9] J. H. Schmidhuber and D. Prelinger. Discovering predictable classifications. *Neural Computation*, 5(4):625–635, 1993.

[10] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[11] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[12] A. Wyner and J. Ziv. Fixed data base version of the Lempel-Ziv data compression algorithm. *IEEE Transactions Information Theory*, 37:878–880, 1991.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(5):337–343, 1977.

LIST OF TABLE CAPTIONS

Method	Av. compression ratio	Variance
Huffman Coding (UNIX: pack)	1.74	0.0002
Lempel-Ziv Coding (UNIX: compress)	1.99	0.0014
METHOD 3, $n = 5$	2.20	0.0014
Improved Lempel-Ziv (UNIX: gzip -9)	2.29	0.0033
METHOD 1, $n = 5$	2.70	0.0158
METHOD 2, $n = 5$	2.72	0.0234

Table 1:

Method	Av. compression ratio	Variance
Huffman Coding (UNIX: pack)	1.67	0.0003
Lempel-Ziv Coding (UNIX: compress)	1.71	0.0036
METHOD 3, $n = 5$	1.99	0.0013
Improved Lempel-Ziv (UNIX: gzip -9)	2.03	0.0099
METHOD 1, $n = 5$	2.25	0.0077
METHOD 2, $n = 5$	2.20	0.0112

Table 2: