

Fast Text Compression with Neural Networks

Matthew V. Mahoney

Florida Institute of Technology
150 W. University Blvd.
Melbourne FL 32901
mmahoney@cs.fit.edu

Abstract

Neural networks have the potential to extend data compression algorithms beyond the character level n-gram models now in use, but have usually been avoided because they are too slow to be practical. We introduce a model that produces better compression than popular Lempel-Ziv compressors (zip, gzip, compress), and is competitive in time, space, and compression ratio with PPM and Burrows-Wheeler algorithms, currently the best known. The compressor, a bit-level predictive arithmetic encoder using a 2 layer, 4×10^6 by 1 network, is fast (about 10^4 characters/second) because only 4-5 connections are simultaneously active and because it uses a variable learning rate optimized for one-pass training.

Keywords: Neural networks, Text compression, Data compression, On-line training/learning, Maximum entropy, Prediction, Efficiency.

1. Introduction

One of the motivations for using neural networks for data compression is that they excel in complex pattern recognition. Standard compression algorithms, such as Lempel-Ziv or PPM (Bell, Witten, and Cleary, 1989) or Burrows-Wheeler (Burrows and Wheeler, 1994) are based on simple n-gram models: they exploit the nonuniform distribution of text sequences found in most data. For example, the character trigram *the* is more common than *qzv* in English text, so the former would be assigned a shorter code. However, there are other types of learnable redundancies that cannot be modeled using n-gram frequencies. For example, Rosenfeld (1996) combined word trigrams with semantic associations, such as “*fire...heat*”, where certain pairs of words are likely to occur near each other but the intervening text may vary, to achieve an unsurpassed word perplexity of 68, or about 1.23 bits per character (bpc)¹, on the 38 million word Wall Street Journal

¹ Rosenfeld’s perplexity of 68 is equivalent to an entropy of $\log_2 68 = 6.09$ bits per word, or 1.11 bpc at 5.5 characters per word (including spaces). In addition, we assume that the 2.2% of the words outside a 20,000 word vocabulary that Rosenfeld modeled as a single token could be encoded at 3 bits per 10 character word using a reasonable n-gram character model, adding $3 \times 0.022 \times 10/5.5 = 0.12$ bpc. The 20K dictionary would add negligible space.

corpus. Connectionist neural models (Feldman and Ballard, 1982) are well suited for modeling language constraints such as these, e.g. by using neurons to represent letters and words, and connections to model associations.

We follow the approach of Schmidhuber and Heil (1996) of using neural network prediction followed by arithmetic encoding, a model that they derived from PPM (prediction by partial match), developed by Bell, Witten, and Cleary. In a predictive encoder, a predictor, observing a stream of input characters, assigns a probability distribution for the next character. An arithmetic encoder, starting with the real interval $[0, 1]$, repeatedly subdivides this range for each character in proportion to the predicted distribution, with the largest subintervals for the most likely characters. Then after the character is observed, the corresponding subinterval becomes the new (smaller) range. The encoder output is the shortest number that can be expressed as a binary fraction within the resulting final range. Since arithmetic encoding is optimal within one bit of the Shannon limit of $\log_2 1/P(x)$ bits (where $P(x)$ is the probability of the entire input string, x), compression ratio depends almost entirely on the predictor.

Schmidhuber and Heil replaced the PPM predictor (which matches the context of the last few characters to previous occurrences in the input) with a 3-layer neural network trained by back propagation to assign character probabilities when given the context as input. Unfortunately the algorithm was too slow to make it practical to test on standard benchmarks, such as the Calgary corpus (Bell, Witten, and Cleary, 1989). Training on a 10K to 20K text file required days of computation on an HP 700 workstation, and the prediction phase (which compressed English and German newspaper articles to 2.94 bpc) ran 1000 times slower than standard methods. In contrast, the 2-layer network we describe, which learns and predicts in a single pass, will compress a similar quantity of text in about 2 seconds on a 100 MHz 80486 (typically achieving better compression).

In section 2, we describe a neural network that predicts the input one bit at a time, and show that it is equivalent to the maximum-entropy (Rosenfeld, 1996; Manning and Schütze, 1999, pp. 589 ff.) approach to combining prob-

abilistic constraints. In (3), we derive an optimal one-pass learning rate for independent, stationary inputs, and extend the model to the more general case. In (4), we compare this model to a number of variations and find experimentally that it is the best. In (5), we find experimentally that neural network compression performs almost as well as PPM and Burrows-Wheeler (current the best known character models) in both speed and compression ratio, and better (but slower) than popular Lempel-Ziv models such as UNIX compress, zip, and gzip. In (6) we describe the implementation.

2. A Maximum Entropy Neural Network

In a predictive encoder, we are interested in predicting the next input symbol y , given a feature vector $\mathbf{x} = x_1, x_2, \dots, x_M$, where each x_i is a 1 if a particular context is present in the input history, and 0 otherwise. For instance, we may wish to find $P(y = \text{elth})$, the probability that the next symbol is e in context th , i.e. $x_{th} = 1$ and all other $x_i = 0$.

In our implementation, we predict one bit at a time, so y is either 0 or 1. We can estimate $P(y = 1|x_i) \approx N(1)/N$, for each context x_i , by counting the number of times, $N(1)$, that $y = 1$ occurs in the N occurrences of that context. In an n -gram character model (n about 4 to 6), there would be one active input ($x_i = 1$) for each of the n contexts of length 0 to $n - 1$. (In practice, the length 0 context was not used).

The set of known probabilities $P(y|x_i)$ does not completely constrain the joint distribution $P(y|\mathbf{x})$ that we are interested in finding. According to the maximum entropy principle, the most likely distribution for $P(y|\mathbf{x})$ is the one with the highest entropy, and furthermore it must have the form (using the notation of Manning and Schütze)

$$P(\mathbf{x}, y) = 1/Z \prod_i \alpha_i^{f_i(\mathbf{x}, y)} \quad (1)$$

where $f_i(\mathbf{x}, y)$ is an arbitrary ‘‘feature’’ function, equal to x_i in our case, α_i are parameters to be determined, and Z is a normalization constant to make the probabilities sum to 1. The α_i are found by *generalized iterative scaling* (GIS), which is guaranteed to converge to the unique solution. Essentially, GIS adjusts the α_i until $P(\mathbf{x}, y)$ is consistent with the known probabilities $P(x_i, y)$ found by counting n -grams.

Taking the log of (1), and using the fact that $P(y|\mathbf{x}) = P(\mathbf{x}, y)/P(\mathbf{x})$, we can rewrite (1) in the following form:

$$P(y|\mathbf{x}) = 1/Z' \exp(\sum_i w_i x_i) \quad (2)$$

where $w_i = \log \alpha_i$. Setting Z' so that $P(y = 0|\mathbf{x}) + P(y = 1|\mathbf{x}) = 1$, we obtain

$$P(y|\mathbf{x}) = g(\sum_i w_i x_i), \text{ where} \quad (3)$$

$$g(x) = 1/(1 + e^{-x}) \quad (4)$$

which is a 2-layer neural network with M input units x_i and a single output $P(y = 1|\mathbf{x})$.

Analogous to GIS, we train the network to satisfy the known probabilities $P(y|x_i)$ by iteratively adjusting the weights w_i to minimize the error, $E = y - P(y)$, the difference between the expected output y (the next bit to be observed), and the prediction $P(y)$.

$$w_i = w_i + \Delta w_i \quad (5)$$

$$\Delta w_i = \eta x_i E \quad (6)$$

where η is the learning rate, usually set *ad hoc* to around 0.1 to 0.5. This is fine for batch mode training, but for on-line compression, where each training sample is presented only once, we need to choose η more carefully.

3. Maximum Entropy On-line Learning

We now wish to find the learning rate, η , required to maintain the maximum-entropy solution after each update. We consider first the case of independent inputs ($P(x_i, x_j) = P(x_i)P(x_j)$), and a stationary source ($P(y|x_i)$ does not vary over time). In this case, we could just compute the weights directly. If we let $x_i = 1$ and all other inputs be 0, then we have:

$$p \equiv P(y|x_i = 1) = g(w_i) \quad (7)$$

$$w_i = g^{-1}(p) = \ln p/(1 - p) \approx \ln N(1)/N(0) \quad (8)$$

where $N(1)$ and $N(0)$ are the number of times y has been observed to be a 1 or 0 respectively in context x_i . (The approximation holds when the $N(\bullet)$ are large. We will return to the problem of zero counts later).

If we observe $y = 1$, then w_i is updated as follows:

$$\Delta w_i = \ln(N(1)+1)/N(0) - \ln N(1)/N(0) \approx 1/N(1) \quad (9)$$

and if $y = 0$,

$$\Delta w_i = \ln N(1)/(N(0)+1) - \ln N(1)/N(0) \approx -1/N(0) \quad (10)$$

We can now find η such that the learning equation $\Delta w_i = \eta x_i E$ (which converges whether or not the inputs are independent) is optimal for independent inputs. For the case $x_i = 1$,

$$\Delta w_i = \Delta w_i E/E = \eta E \quad (11)$$

$$\eta = \Delta w_i/E \quad (12)$$

If $y = 1$, then

$$E = 1 - p = 1 - N(1)/N = N(0)/N \quad (13)$$

$$\eta = \Delta w_i/E = 1/EN(1) = N/N(0)N(1) = 1/\sigma^2 \quad (14)$$

And if $y = 0$, then

$$E = -p = -N(1)/N \quad (15)$$

$$\eta = \Delta w_i/E = -1/EN(0) = N/N(0)N(1) = 1/\sigma^2 \quad (16)$$

where $N = N(0) + N(1)$, and $\sigma^2 = N(0)N(1)/N = Np(1 - p)$ is the variance of the training data.

If the inputs are correlated, then using the output error to adjust the weights still allows the network to converge, but not at the optimal rate. In the extreme case, if there are m perfectly correlated copies of x_i , then the total effect will

be equal to a single input with weight mw_i . The optimal learning rate will be $\eta = 1/m\sigma^2$ in this case. Since the correlation cannot be determined locally, we introduce a parameter η_L , called the long term learning rate, where $1/m \leq \eta_L \leq 1$. The weight update equation then becomes

$$\Delta w_i = \eta_L E / \sigma^2 \quad (17)$$

In addition, we have assumed that the data is stationary, that $p = P(y|x_i)$ does not change over time. The following bit stream fragment of y for one particular context x_i suggests that a better model is one where p periodically varies between 0 and 1 over time. The input is `alice30.txt`, *Alice in Wonderland* by Lewis Carroll, from Project Gutenberg. The context $x_i = 1$ for 256 particular values (selected by a hash function) of the last 24 bits of input.

```
0000000000000000000010000000000000000000000001111111110
000000000000000000000000000000000000000000000111111111
11111111111111111111000000000000000000000000000000001000
000000000000000000001111110100010100010100010100010100010
100010100010100010100010100010100010100010100010100010100
01010001010001010001010001100000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
011111111111111111110000010000000010000000100000100000
001000000000000000000000000000000000000000000000000000000000
000001000001000000001000000000000000000000000000000000000000
```

If the probability were fixed, we would not expect to find long strings of both 0's and 1's. It is clear from this example that the last few bits of input history is a better predictor of the next bit than simply weighting all of the bits equally, as η_L does. The solution is to impose a lower bound, η_S , on the learning rate, a parameter that we call the *short term* learning rate. This has the effect of weighting the last $1/\eta_S$ bits more heavily.

$$\Delta w_i = (\eta_S + \eta_L / \sigma^2) E \quad (18)$$

Finally, recall that $\sigma^2 = N(0)N(1)/N = Np(1 - p)$, which means that the learning rate is undefined when either count is 0, or equivalently, when $p = N(1)/N$ is 0 or 1. A common solution is to add a small offset, d , to each of the counts $N(0)$ and $N(1)$. For the case $d = 1$, we have Laplace's law of succession, which is optimal when all values of p are *a priori* equally likely (Witten and Bell, 1991). Cleary and Teahan (1995) found, however, that the optimal value of d depends on the data source. For text, their experiments suggest a value of $d \approx 0.1$, though we found that 0.5 works well. Thus, in (18), we use

$$\sigma^2 = (N+2d) / (N(0)+d)(N(1)+d) \quad (19)$$

4. Alternative Approaches

So far we have only shown how to optimally predict stationary data with independent features, and suggested extensions to the more general case. To test these, a large number of alternatives (about 20) were tested experimen-

tally. Space prevents us from describing all but three of the most promising approaches, none of which improved over the methods described.

For all of the tests, the input was *Alice in Wonderland*, 152,141 bytes. A neural network was constructed with 258K input units, divided into four sets of contexts, listed below. Exactly one input from each set was active at any time. The context sets were:

- The last 8 bits of input, plus the position (0-7) within the current byte, for a total of 2^{11} contexts. Thus, unit x_i is active if the last 8 bits of input, concatenated with the 3-bit number representing the bit position 0-7, is exactly equal to the 11-bit number i .
- The last 16 bits of input (2^{16} contexts).
- The last 24 bits of input, hashed to a 16 bit value (2^{16} contexts). The hash function is a polynomial function of the input bits, modulo 2^{16} , chosen mainly for speed).
- The last 32 bits of input, hashed to a 17 bit value (2^{17} contexts).

These contexts are similar to the ones used by PPM compressors, which typically use the last 1, 2, 3, and 4 characters to predict the next character.

The parameters of (18) and (19) were hand tuned to $\eta_L = 0.5$, $\eta_S = 0.08$, and $d = 0.5$, to optimize compression, obtaining a baseline of 2.301 bpc. This compares to 2.466 bpc for conventional training (η_S tuned to 0.375 with η_L fixed at 0), 2.368 bpc for the assumption of stationary, independent inputs, ($\eta_L = 1$, $\eta_S = 0$), and 2.330 bpc allowing for some correlation between inputs, (η_L tuned to 0.6-0.7 with η_S fixed at 0).

4.1. Adaptive learning rate

Of the many alternatives tested, the most successful approach was to adjust the learning rate dynamically for each connection. The idea is to observe the last two errors when $x_i = 1$, and adjust the learning rate in the direction that would have reduced the second error.

$$\begin{aligned} \Delta w_i &= E \eta_i \\ \eta_i &:= (A E E_i + 1) \eta_i \\ E_i &= E \end{aligned}$$

where E and E_i are the current and previous errors, and $A < 1$ is a parameter that determines how rapidly the learning rate is adjusted. This works fairly well. After hand tuning, compression of 2.344 bpc (0.043 over baseline) was obtained with $A = 0.5$, and η_i initially 0.5.

4.2. Rate decay

The second best alternative tried was to take a more direct approach to decreasing the learning rate each time x_i is active, which (18) does indirectly:

$\Delta w_i = \eta_i E$, where

η_i is updated by $\eta_i := A\eta_i + B$.

The result was 2.380 bpc (0.079 over baseline), only a little better than the best fixed rate (2.466 bpc). This was obtained by tuning A to 1/2, B to 1/8, and η_i to an initial value of 1.5. With these parameters, η_i converges to $B/(1 - A) = 1/4$.

4.3. PPM and interpolation

In 4.1 and 4.2, we experimented with different learning rates. Now we experiment with different methods of combining the known probabilities $P(y|x_i)$. The most common alternatives to maximum entropy are linear interpolation and PPM. Of the $m = 4$ active contexts with lengths 11, 16, 24, and 32 bits, PPM simply selects the longest one which occurred at least once before ($N > 0$), and estimates $P(y) = (N(1) + d)/(N + 2d)$. In contrast to larger alphabets, binary PPM performs surprisingly poorly:

d	bpc
0.1	3.930
0.25	3.798
0.5	3.800
1 (Laplacian)	3.915

Table 1. Compression of *Alice in Wonderland* using binary PPM.

Binary PPM compression was much worse than fixed rate learning (2.445 bpc), and worse than the 3.123 bpc obtained by simply averaging the four probabilities (with $d = 0.5$). It was thought that weighted averaging, with weights proportional to set size (i.e. 2^{11} , 2^{16} , 2^{24} , 2^{32}) to favor the longer and presumably more reliable contexts, should improve compression over simple averaging. Instead, compression worsened to 3.182 bpc.

5. Experimental Evaluation

The small (258K input) neural network of section 4, and a larger network with 2^{22} (about 4×10^6) inputs were compared to seven popular and top rated compression programs. Compression ratios and times for *Alice in Wonderland* and ratios for *book1* from the Calgary corpus (Hardy’s *Far from the Madding Crowd*, 768,771 bytes) are shown below. Parameters for both neural networks were hand tuned only for *Alice in Wonderland*, and then tested on *book1* without further tuning. Tests were performed on a 100 MHz 80486 with 64 MB of RAM under Windows 95.

Program	Ver.	Year	Type	Alice bpc	Comp (sec)	De- comp bpc	Book1 bpc
compress	4.3d	1990	LZ	3.270	1	0.5	3.486
pkzip	2.04e	1993	LZ	2.884	6	<0.5	3.288
gzip -9	1.2.4	1993	LZ	2.848	7	0.5	3.250
szip -b41 -o0	1.05Xf	1998	BW	2.239	9	6	2.345
ha a2	0.98	1993	PPM	2.171	16	16	2.453
boa -m15	0.58b	1998	PPM	2.061	33	34	2.204
rkive -mt3	1.91b1	1998	PPM	2.055	134	117	2.120
neural small	P5	1999	NN	2.301	24	24	2.508
neural large	P6	1999	NN	2.129	26	26	2.283

Table 2. Compression ratios and times for *Alice in Wonderland* and ratios for *book1*²

The large model uses 5 sets of inputs. In each set, the index i of the active input x_i is a 22-bit hash of the last n (1 to 5) characters, the last 0 to 7 bits of the current character, and the position (0 to 7) within the current character. For $n = 1$, there are only 64K possible values. The other four sets ($n = 2$ through 5) overlap each other in a space of size $2^{22} - 64K = 4,128,768$. The parameters were hand tuned on *Alice in Wonderland* to $\eta_L = 0.35$ and $\eta_S = (0.08, 0.15, 0.2, 0.2, 0.2)$ for the context sets for $n = (1, 2, 3, 4, 5)$, with $d = 0.5$.

The seven compression program options were set for maximum text compression. UNIX *compress*, *pkzip*, and *gzip* are popular Lempel-Ziv compressors because of their speed, and in spite of their poor compression. They work by replacing reoccurring n -grams with a pointer to the previous occurrence.

Gilchrist (1999) rated *boa* (Sutton, 1998) as giving the best compression from among hundreds of archivers on the Calgary corpus, and *rkive* (Taylor, 1998) as the best on the text portion as of Sept. 1998. Both are PPM programs, which predict characters based on n -gram counts. Bell (1998) rated *szip* (Schindler, 1998), a Burrows-Wheeler encoder, the best on the large (2-4 MB) text portion of the Canterbury corpus. It sorts the input stream by context, then uses an adaptive 1-gram compressor on the permuted stream. *ha* (Hirvola, 1993) is an early PPM encoder, rated the best compressor in 1994 (Data Compression FAQ, 1998).

The large model compressed to within 0.074 bpc of *rkive*, the best compressor on *Alice in Wonderland*, and 0.719 bpc better than *gzip*, the best LZ compressor. With no further tuning, it compressed within 0.163 bpc of *rkive* on *book1*, and 0.967 bpc better than *gzip*.

In another test, the large model was compared with the top-rated *boa* on the Calgary corpus of 14 text and binary

² *rkive* ran out of memory using the -mt3 option on *book1*, so -mt2 was used. The options -mt0 through -mt3 produce successively better compression at the cost of memory.

files, again with no further tuning. Both programs use about the same amount of memory (15 MB for *boa*, 16 MB for the neural network), and both compress in “solid” mode (*boa* option *-s*), treating the input as a single file. The result, using the same file order, was 1.904 bpc for *boa* in 720 seconds, and 2.142 bpc for the neural network in 537 seconds.

6. Implementation

Both the large and small neural networks use 4 bytes of memory for each input. The weight w_i is represented as a 16 bit signed number with a 10 bit fraction, i.e., ± 32 with precision 0.001. The counts $N(0)$ and $N(1)$ are represented as 8 bit unsigned numbers with a 1 bit fraction, i.e. 0 to 127.5 (integers 0 to 255 in the large model). To prevent overflow, both counts are divided by 2 whenever either count exceeds 125 (250 in the large model). This has the same effect as a small η_S , of favoring recent training data. In fact, when $\eta_S = 0$, rolling over the counts at around 25 or 30 instead of 125 improved the compression.

The arithmetic encoder stores the current range as a pair of unsigned 32 bit numbers, writing out the leading bytes whenever they match. This, and using a 16 bit unsigned probability, results in a slight loss of efficiency, experimentally less than 0.0001 bpc.

Additional optimization techniques include the exclusive use of integer arithmetic, using bit shifts in place of division when possible, using table lookup to compute $g(\bullet)$, and representing the weights as an array of structures (instead of a structure of arrays) to optimize cache usage. The programs were written in C++.

7. Conclusions

It was shown that it is practical to use neural networks for text compression, an application requiring high speed. Among neural models, the best one found combines long and short term learning rates to achieve a balance between using the entire input history and favoring the most recent data to adapt to changing statistics..

Although we optimized the neural network to predict text, it was found to give good performance on a wide variety of data types. It would probably be better to have the program automatically adjust η_S and η_L , rather than setting them *ad hoc*, but this would just introduce another parameter to determine the adjustment rate. The problem of *ad hoc* parameters seems to be unavoidable.

Research is ongoing, and it is expected that improved compressors will be found. We have yet to investigate models that exploit syntactic or semantic constraints, our reason for using neural networks in the first place. Improved models (C++ source code and Windows 95 execu-

tables) will be posted to <http://cs.fit.edu/~mmahoney/compression> as they become available.

Acknowledgments

I would like to thank Dr. Phil Chan for helpful comments on this paper.

References

- Bell, Timothy, Witten, Ian H., and Cleary, John G. 1989. Modeling for Text Compression, *ACM Computing Surveys* 21(4): 557-591.
- Bell, Timothy. 1998. Canterbury Corpus, <http://corpus.canterbury.ac.nz>
- Burrows, M., and Wheeler, D. J. 1994. A Block-Sorting Lossless Data Compression Algorithm. Digital Systems Research Center, <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html>
- Cleary, John G., and Teahan, W. J. 1995. Experiments on the Zero Frequency Problem. *Proc. Data Compression Conference*: 480.
- Data Compression FAQ 1998. <http://www.cs.ruu.nl/wais/html/na-dir/compression-faq/html>
- Feldman, J. A., and Ballard, D. H. 1982. Connectionist Models and their Properties, *Cognitive Science* 6: 205-254
- Gilchrist, Jeff, 1999. Archive Comparison Test, <http://act.by.net/act.html>
- Hirvola, H., 1993, ha 0.98, <http://www.webwaves.com/arcers/msdos/ha098.zip>
- Manning, Christopher D., and Schütze, Hinrich, 1999, *Foundations of Statistical Natural Language Processing*. Cambridge Mass.: MIT Press.
- Rosenfeld, Ronald 1996. A Maximum Entropy Approach to Adaptive Statistical Language Modeling. *Computer, Speech and Language*: 10. See also <http://www.cs.cmu.edu/afs/cs/user/roni/WWW/me-csl-revised.ps>
- Schindler, Michael, 1998. *gzip* homepage, <http://www.compressconsult.com/gzip/>
- Schmidhuber, Jürgen, and Heil, Stefan 1996, Sequential Neural Text Compression, *IEEE Trans. on Neural Networks* 7(1): 142-146.
- Sutton, Ian, 1998, *boa* 0.58 beta, <ftp://ftp.cdrom.com/pub/sac/pack/boa058.zip>
- Taylor, Malcolm, *RKIVE* v1.91 beta 1, 1998, <http://www.geocities.com/SiliconValley/Peaks/9463/rkive.html>
- Witten, Ian H., and Timothy C. Bell, 1991, The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression, *IEEE Trans. on Information Theory*, 37(4): 1085-1094