NEURAL PREDICTORS FOR DETECTING AND REMOVING REDUNDANT INFORMATION

Jürgen Schmidhuber
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland
juergen@idsia.ch - http://www.idsia.ch/~juergen

Abstract

The components of most real-world patterns contain redundant information. However, most pattern classifiers (e.g., statistical classifiers and neural nets) work better if pattern components are nonredundant. I present various unsupervised nonlinear predictor-based "neural" learning algorithms that transform patterns and pattern sequences into less redundant patterns without loss of information. The first part of the paper shows how a neural predictor can be used to remove redundant information from input sequences. Experiments with artificial sequences demonstrate that certain supervised classification techniques can greatly benefit from this kind of unsupervised preprocessing. In the second part of the paper, a neural predictor is used to remove redundant information from natural text. With certain short newspaper articles, the neural method can achieve better compression ratios than the widely used asymptotically optimal Lempel-Ziv string compression algorithm. The third part of the paper shows how a system of co-evolving neural predictors and neural code generating modules can build factorial (statistically nonredundant) codes of pattern ensembles. The method is successfully applied to images of letters randomly presented according to the probabilities of English language.

1 INTRODUCTION

In this paper I will show how neural nets can be used to detect redundant information in input data. Once redundant information is detected, the data can be compressed. This can sometimes greatly simplify and speed up goal directed learning. Three methods for redundancy reduction will be presented. All of them are based on unsupervised networks that learn to "predict away" redundant information.

1.1 WHAT IS REDUNDANT INFORMATION?

The statistician's point of view is this: Suppose we draw input patterns from an m-dimensional distribution given by a random variable $X = (X_1, X_2, \ldots, X_m)$. The p-th pattern is a vector $x^p = (x_1^p, x_2^p, \ldots, x_m^p)^T \in R^m$.

Redundancy means that pattern components share mutual information. It means that if we know certain components of some pattern, then we already know something about other components. Redundancy implies statistical dependence of the components:

$$P(X_i = x_i) \neq P(X_i = x_i \mid \{X_k = x_k, k \neq i\}).$$

1.2 WHAT IS REDUNDANCY REDUCTION?

With a given set of patterns, the goal of redundancy reduction without loss of information is this: Generate a new set of pattern vectors (the code) such that (a) for each pattern x^p from the original set there is exactly one pattern y^p from the code (y^p *represents* x^p) and (b) the components of code patterns are "less dependent on each other". To make (b) more precise, let us measure the degree of the redundancy of a random variable X with components X_i by

$$R(X) = \frac{\sum_i H(X_i) - H(X)}{H(X)},$$

where

$$H(Z) = -\sum_{all\ cases\ \alpha} P(Z = \alpha) log P(Z = \alpha)$$

denotes the entropy of variable Z. Ideally, the result of redundancy reduction is a "factorial code". With a factorial code, the code components are statistically independent:

$$\forall p: \quad P(x^p) = P(y^p) = \prod_i P(y_i = y_i^p)$$

(throughout this paper, the subscript i of a symbol representing a vector indexes the i-th component of the vector).

1.3 REDUNDANCY REDUCTION: WHY?

This paper sequentially addresses three reasons for redundancy reduction:

- 1. *Redundancy reduction can help to decrease the search space for goal directed learning procedures.* The next section will show this in the context of sequence classification, where redundancy reduction can sometimes help to achieve enormous learning speed-ups in comparison to more traditional approaches.

- 2. *Redundancy reduction allows for data compression.* Section 3 will review a "neural" method for text compression. With certain short newspaper articles, the neural method can achieve better compression ratios than the widely used asymptotically optimal Lempel-Ziv string compression algorithm (as embodied by the UNIX functions "compress" and "gzip").
- 3. *Redundancy reduction promises to simplify statistical classifiers.* For efficiency reasons, most statistical classifiers (e.g. Bayesian pattern classifiers) assume statistical independence of their input variables (corresponding to the pattern components). We would like to have a method that takes an arbitrary pattern ensemble as an input and generates an equivalent factorial code. The code (instead of the original ensemble) could be fed into an efficient conventional classifier, which in turn could achieve its optimal performance. Section 4 will review a "neural" method designed to generate (nearly) factorial binary codes.

1.4 REDUNDANCY REDUCTION: HOW?

All the methods in the following sections will be instances of the following quite general scheme:

- **1. Measure redundancy by adaptive neural predictors.**
- **2. Remove redundant information found by the predictors.**

The next section gives the first example.

2 EXAMPLE 1: Sequence Classification

Outline. This section starts by describing a sequence classification task. The training sequences consist of relatively nonredundant components. It is shown how to solve the task with a conventional recurrent neural net. Then it is shown that the recurrent net fails to learn a very similar task involving training sequences conveying a lot of redundant information. Finally it is demonstrated how unsupervised adaptive redundancy reduction allows for learning the second task, too.

2.1 THE EASY TASK

Figure 1 shows a stochastic automaton for generating sequences. To produce a string of symbols, we begin in the start state. From there we move to state 1 and emit the symbol "a". In state 1 there is a choice of three alternatives: We can go to state 1 again and emit another "a". We can go to state 2 and emit a "b". We can go to state 3 and emit an "e". Whenever there are alternatives, the automaton selects among them with equal probability. States 4 and 5 are

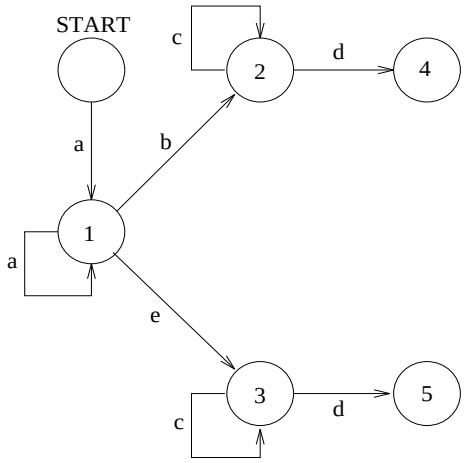

Figure 1: A stochastic automaton for generating symbol sequences.

final states and correspond to sequence endings. We say that a sequence that ends in state 4 is of class 1. We say that a sequence that ends in state 5 is of class 2. Here are typical sequences of class 1: *abcd, aaaaaaabcd, aaabccccccd.* Here is a typical sequence of class 2: *aaaeccccccd.*

The goal for a (yet unspecified) sequence classifier is to read sequences, one symbol at a time, and to learn to classify them with respect to whether they belong to class 1 or class 2. The classifier does not know anything about the automaton. All it sees are the successive symbols and a teacher signal at the end of each sequence. The teacher signal provides the information about the desired classification. Consider the last two sequences examples above. They illustrate that the difference between class 1 and class 2 only depends on the first letter in the sequence that is not an "a". If this letter is a "b", then the sequence is of class 1. If this letter is an "e", then the sequence is of class 2. Therefore, the classifier somehow has to learn to store the occurrence of the first letter that is not an "a" and to memorize it for an unknown number of discrete time steps – until the end of the sequence is reached. Otherwise it will not be able to classify correctly.

2.1.1 SIMULATION

We use local input representation. A feedforward neural net cannot learn the problem, due to the arbitrary time lags that may occur. Instead, we need something like a recurrent net (e.g. [18] [37] [17] [40] [15] [39] [38] [20] [7]).

A conventional fully recurrent net with 5 input units, 5 hidden units, 1 output unit, 1 "bias unit", and a learning rate of 0.5, needed less than 10000 training sequences to come up with correct classifications in 100% of all test

sequences. (class 1: output activation > 0.9, class 2: output activation < 0.1 at the end of a sequence).

This sets the stage for the more difficult task described in the next subsection. It involves highly redundant sequences and will be harder to learn.

2.2 THE DIFFICULT TASK

The information conveyed by the sequence source (the automaton above) can be measured in terms of the entropy of the probability distribution of the possible sequences: $-\sum_{all\ possible\ sequences\ p} P(p) log P(p)$. Our next step will be to modify the automaton such that its entropy remains the same but the redundancy among the sequence components increases. All transitions of the form $i \rightarrow^x j$ (read: "go from state i to state j and emit symbol x") are replaced by a subautomaton consisting of the deterministic sequence of transitions $i \rightarrow^x i_2 \rightarrow^x i_3 \rightarrow^x \ldots \rightarrow^x i_{20} \rightarrow^x j$. The modified automaton generates redundant sequences like this one (from class 2):
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaeeeeeeeeeeeeeeeeeeee
ccccccccccccccccccccccccccccccccccccccccdddddddddddddddddddd.
This sequence is redundant because many of its components are predictable from other components.

2.2.1 SIMULATION

With the modified automaton, it turns out that conventional recurrent networks **fail to learn** the correct classifications within 10^7 training sequences (various topologies were tested). Two related reasons are:

- Time lags are too long (error signals become less significant while moving "back into time"). See, e.g., [7].
- The presumed search space is huge (in principle, the recurrent net considers all possible symbol combinations as equal candidates for being the reason for the final classification).

But the "real" search space ought to be small, because most possible symbol combinations can never occur. The modification of the automaton did not cause a change in entropy. How can an adaptive system find this out? The next section gives an answer.

2.3 REDUNDANCY REDUCTION FOR SEQUENCES

This section shows how to "predict away" redundant information in sequences constructed from a finite set of possible input symbols. 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 other words, we can forget about the predictable input vectors. We need to look at the unpredictable inputs only. Only the unexpected deserves attention [21]. We apply this insight to sequences generated by the modified automaton above.

2.3.1 SIMULATION

We start with 100 training sequences for a recurrent predictor network with 5 input units, 5 output units, 1 bias unit, and a learning rate of 0.5. The predictor sees the original sequences, one symbol at a time. At every time step, it tries to predict the next element (the symbol whose input representation is closest to the predictor's real-valued output vector is taken as the deterministic prediction).

Then we "freeze" the predictor (never change its weights again). Now training sequences for a separate recurrent classifier with 6 input units, 5 hidden units, 1 bias unit, and learning rate 0.5, are "filtered through" the predictor: The classifier sequentially is fed with the more compact sequence descriptions consisting of the unexpected inputs only. Each input is concatenated with a representation of the time that went by since the last unexpected input (the exponentially decaying activation of a special input unit is reset to 1 whenever there is an unexpected input). The compact sequence conveys all the information conveyed by the original sequence, **and the classifier learns the correct classifications within less than 10000 training sequences.**

Recall that conventional recurrent networks failed to learn the task within 10^7 training sequences. In this special case, the speed-up factor obtained by adaptive redundancy reduction is at least 10^3.

More details and extensions of the principle above can be found in [21], [23], [29], and especially in [24]. The next section describes a somewhat different kind of neural predictor for compressing natural text (as opposed to artificial symbol strings).

3 EXAMPLE 2: Text Compression

The example from the previous section was based on artificial data from a stochastic automaton. Can neural predictors offer something for redundancy

reduction in natural language? How do they compare to standard data compression algorithms?

The method [28] reviewed in this section is an instance of a strategy known as "predictive coding" or "model-based coding". A neural predictor network P is trained to approximate the conditional probability distribution of possible characters, given the previous characters. P's outputs are fed into the Arithmetic Coding algorithm (e.g. [41]) that generates 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) [28].

3.1 PREDICTING CONDITIONAL PROBABILITIES

With the *offline* variant of the approach, P's training phase is based on a set F of training files. 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_m^f.

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

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

where $\circ$ is the concatenation operator for vectors. P produces as an output P_m^f, a k-dimensional output vector. Using back-propagation [36][9][16][19], P is trained to minimize

$$\frac{1}{2} \sum_{f \in F} \sum_{m > n} || \, r(c_m^f) - P_m^f \, ||^2 \, .$$

Let $(P_m^f)_j$ denote the j-th component of the vector P_m^f. Due to the local character representation, this error function is minimized if, for all f and for all appropriate $m > n$, $(P_m^f)_i$ is equal to the conditional probability

$$P(c_m^f = z_i \mid c_{m-n}^f, \ldots, c_{m-1}^f).$$

For normalization purposes, we define

$$P_m^f(i) = \frac{(P_m^f)_i}{\sum_{j=1}^k (P_m^f)_j}.$$

3.2 USING THE PREDICTOR FOR COMPRESSION

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_m^f $(m > n)$, the predictor emits its output P_m^f based on the n previous characters. There will be a k such that $c_m^f = z_k$. The estimate of $P(c_m^f = z_k \mid c_{m-n}^f, \ldots, c_{m-1}^f)$ is given by $P_m^f(k)$. The code of c_m^f, the bitstring $code(c_m^f)$, is generated by feeding the probability estimates of the predictor into the Arithmetic Coding algorithm (see e.g. [41]). $code(c_m^f)$ is written into the compressed file.

The information in the compressed file is sufficient for reconstructing the original file. This is done with the "uncompress" algorithm, which works as follows: Again, for each character c_m^f $(m > n)$, the predictor (sequentially) emits its output P_m^f based on the n previous characters, where the c_l^f with $n < l < m$ were gained sequentially by feeding the approximations $P_l^f(k)$ of the probabilities $P(c_l^f = z_k \mid c_{l-n}^f, \ldots, c_{l-1}^f)$ into the inverse Arithmetic Coding procedure (e.g. [41]). The latter is able to correctly decode c_l^f from $code(c_l^f)$. In other words, to correctly decode some character, we first need to decode all previous characters.

3.3 SIMULATIONS

Our current computing environment prohibits extensive experimental evaluations of the method above. The predictor updates turn out to be quite time consuming, which makes special neural net hardware recommendable. The limited software simulations presented in this section, however, will show that the "neural" compression technique can achieve "excellent" compression ratios. Here the term "excellent" is defined by a statement from [5]:

> "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."

Here the average compression ratio is the average ratio between the lengths of original and compressed files.

In collaboration with Stefan Heil (a former student at TUM), the method was applied to German newspaper articles [27, 28]. The results were compared 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. [5]), while "compress" and "gzip" are based on the Lempel-Ziv technique [43]. As the file size goes to infinity, Lempel-Ziv becomes *asymptotically optimal* in a certain information theoretic sense [42].

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

Method	Compression Ratio
Huffman Coding (UNIX: pack)	1.41
Lempel-Ziv Coding (UNIX: compress)	1.98
Improved Lempel-Ziv (UNIX: gzip -9)	2.25
"Neural" method , $n = 5$	2.68

Table 1: *Compression ratios of various compression algorithms for short German text files (*$<$ 20000 *Bytes) from the unknown test set.*

upper case and lower case letters, digits, interpunction symbols, and special German letters like "ö", "ü", "ä".

The test set consisted of 10 newspaper articles excluded from the training set, each containing between 10000 and 20000 characters. Table 1 lists the average compression ratios. The "neural" method outperformed the strongest conventional competitor, the UNIX "gzip" function based on the asymptotically optimal Lempel-Ziv algorithm.

Note that the time-window was quite small ($n = 5$). 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.

3.4 ON-LINE METHODS / LIMITATIONS

A disadvantage of the offline technique above is that it is offline: the predictor does not adapt to the specific text file it sees. This limitation is not essential, however. It is straight-forward to construct an *online* variant of the approach. With the online variant, the predictor continues to learn *during* compression. The online 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. *This implies that 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.* Of course, the online method promises higher compression ratios than the offline method.

The main disadvantage of both online and offline variants, however, is their computational complexity. The current offline implementation is clearly slower than conventional standard techniques, by about three orders of magnitude (but no attempt was made to optimize the code with respect to speed). And the complexity of the online method is even worse (the exact slow-down factor

depends on the precise nature of the learning protocol, of course). For this reason, especially the promising online 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.

Among the methods presented in this paper, perhaps the most interesting way of using neural predictors for redundancy reduction is the one described in the next section.

4 EXAMPLE 3: Discovering Factorial Codes

In this section, we go back to the general definition of redundancy in the introduction. There it was mentioned that the ideal of a low-redundancy code is a factorial code (e.g. [1]). Recall that with a factorial code, the pattern components of the code vectors y^p (representing input patterns x^p) are statistically independent:

$$\forall p: \quad P(x^p) = P(y^p) = \prod_i P(y_i = y_i^p). \tag{1}$$

The next section presents a predictor-based method designed to find *binary* factorial codes. With binary factorial codes, equation (1) implies

$$E(y_i \mid \{y_k, k \neq i\}) = E(y_i)$$

for all i (here E denotes the expectation operator).

4.1 PREDICTABILITY MINIMIZATION

In its most simple form, predictability minimization is based on a feedforward network with m input units and n output units (or code units). The i-th code unit produces an output value $y_i^p \in [0, 1]$ in response to the current external input vector x^p. The central idea is: **For each code unit there is an adaptive predictor network that tries to predict the code unit from the remaining $n-1$ code units. But each code unit in turn tries to become as unpredictable as possible.** The only way it can do so is by representing environmental properties that are statistically independent from environmental properties represented by the remaining code units. The principle of predictability minimization was first described in [22].

The predictor network for code unit i is called P_i. Its output in response to the $\{y_k^p, \ k \neq i\}$ is called P_i^p. P_i is trained to *minimize*

$$\sum_p (P_i^p - y_i^p)^2, \tag{2}$$

thus learning to predict the the conditional expectation $E(y_i \mid \{y_k, k \neq i\})$ of y_i, given the set $\{y_k, \ \ k \neq i\}$. But the code units try to *maximize* the same (!) objective function the predictors try to minimize:

$$V_C = \sum_{i,p} (P_i^p - y_i^p)^2. \tag{3}$$

Predictors and code units co-evolve by fighting each other. See details in [22] and especially in [24].

Let us assume that the P_i do not get trapped in local minima and perfectly learn the conditional expectations. It then turns out that the objective function V_C (first given in [22]) is essentially equivalent to the following one (also given in [22]):

$$\sum_i VAR(y_i) - \sum_{i,p} (P_i^p - \bar{y}_i)^2, \tag{4}$$

where $\bar{y}_i$ denotes the mean activation of unit i, and VAR denotes the variance operator. The equivalence of (3) and (4) was observed by Peter Dayan, Richard Zemel and Alex Pouget (SALK Institute, 1992). See [24] for details. (4) gives some intuition about what is going on while (3) is maximized. The first term of (4) tends to enforce binary units, while the second term tends to make the conditional expectations equal to the unconditional expectations, thus encouraging statistical independence.

Note that unlike with many previous approaches to unsupervised learning, the system is not limited to local "winner-take-all" representations. Instead, there may be distributed code representations based on many code units that are active simultaneously – as long as the "winners" stand for independent abstract features extracted from the input data. And unlike previous approaches, the method allows for discovering *nonlinear* predictability, and for *nonlinear* pattern transformations to obtain codes with statistically independent components. Note that statistical independence implies decorrelation. But decorrelation does not imply statistical independence.

4.2 SIMULATIONS: IMAGE CODING

In collaboration with Stefanie Lindstädt, the method was applied to a pattern ensemble consisting of 80 characters from the font-courier DEC-dataset. Each character was represented by an image made of 10×15 binary pixels. Since there are only 80 characters but 2^{150} possible patterns that can be represented by 150 pixels, the training set contains an enormous amount of redundant information.

During training, the images were randomly presented according to the probabilities of English language. The unsupervised system had 150 input units, 16 code units, and 1 "bias" unit. Each predictor had 15 input units, 1 "bias" unit, and 1 output unit. The learning rate of the predictors was 10 times as high as the learning rate of the code units. Within 10000 pattern presentations, the

system often learned to generate a loss-free code of the ensemble such that the code was much less redundant than the original data. The redundancy (see the definition in section 1.2) corresponding to the original DEC dataset is 13.41. The redundancy corresponding to a 16-bit code discovered by the system is 2.5. See [14], [13], and [24] for details.

This result corresponds to a dramatic reduction of redundant information, although the achieved value is not optimal. In many realistic cases, however, approximations of nonredundant codes should be satisfactory. It is intended to apply the method to the problem of unsupervised segmentation of real world images. See [30] for an application to simple stereo vision.

One might speculate about whether the brain uses a similar principle based on "code neurons" trying to escape the predictions of "predictor neurons".

5 OUTLOOK

This paper presented a number of possibilities for "neural" redundancy reduction based on Shannon's concept of information [33]. Many potential sources of redundant information have been neglected, however (see, e.g., [6]). Among the things I did not address is the possibility of redundancy among learning strategies [31]. Suppose learning algorithm A is good at solving problems of class $\bar{A}$ but tends to fail with problems of class $\bar{B}$. Suppose learning algorithm B is good at solving problems of class $\bar{B}$ but tends to fail with problems of class $\bar{A}$. But perhaps there is a short algorithm that takes learning algorithm A as an input and outputs learning algorithm B. This implies that there is "algorithmic redundancy" between A and B. Variants of algorithmic redundancy allow for things like "learning by analogy", "learning by chunking", "learning how to learn", etc. [31]. Shannon information, however, is not the right concept to exploit the potential benefits of algorithmic redundancy. Instead we need to look at Kolmogorov complexity or "algorithmic information" [8] [34] [2] [11] [3] and especially at its computationally tractable generalizations (e.g. [4] [10] [12] [35]) to properly treat general (as opposed to conventional statistical) sources of redundant information. This is a recent focus of my research [25, 26, 32, 31].

6 ACKNOWLEDGMENTS

Thanks to Peter Dayan, Richard Zemel and Alex Pouget for sharing their insight regarding the equivalence of equations (4) and (3). Thanks to David MacKay for directing my attention towards Arithmetic Coding. Thanks to Michael C. Mozer for many fruitful discussions. This research was supported in part by a DFG fellowship to the author. I would also like to acknowledge useful comments by Nicol N. Schraudolph sponsored by SNF grant 2100-045700.95/1 "Predictability Minimization".

References

[1] H. B. Barlow, T. P. Kaushal, and G. J. Mitchison. Finding minimum entropy codes. *Neural Computation*, 1(3):412–423, 1989.

[2] G.J. Chaitin. On the length of programs for computing finite binary sequences: statistical considerations. *Journal of the ACM*, 16:145–159, 1969.

[3] G.J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340, 1975.

[4] J. Hartmanis. Generalized Kolmogorov complexity and the structure of feasible computations. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 439–445, 1983.

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

[6] S. Hochreiter and J. Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

[7] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1681–1726, 1997.

[8] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.

[9] Y. LeCun. Une procédure d'apprentissage pour réseau à seuil asymétrique. *Proceedings of Cognitiva 85, Paris*, pages 599–604, 1985.

[10] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[11] L. A. Levin. Laws of information (nongrowth) and aspects of the foundation of probability theory. *Problems of Information Transmission*, 10(3):206–210, 1974.

[12] M. Li and P. M. B. Vitányi. An introduction to Kolmogorov complexity and its applications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 188–254. Elsevier Science Publishers B.V., 1990.

[13] 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.

[14] S. Lindstädt. Comparison of unsupervised neural networks for redundancy reduction, 1993. Master's thesis, Dept. of Comp. Sci., University of Colorado at Boulder.

[15] M. C. Mozer. A focused back-propagation algorithm for temporal sequence recognition. *Complex Systems*, 3:349–381, 1989.

[16] D. B. Parker. Learning-logic. Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT, 1985.

[17] B. A. Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

[18] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

[19] 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.

[20] J. Schmidhuber. A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248, 1992.

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

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

[23] J. 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.

[24] J. Schmidhuber. Netzwerkarchitekturen, Zielfunktionen und Kettenregel. Habilitationsschrift, Institut für Informatik, Technische Universität München, 1993.

[25] J. Schmidhuber. Discovering solutions with low Kolmogorov complexity and high generalization capability. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 488–496. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

[26] J. Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.

[27] J. Schmidhuber and S. Heil. Predictive coding with neural nets: Application to text compression. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1047 – 1054. MIT Press, Cambridge MA, 1995.

[28] J. Schmidhuber and S. Heil. Sequential neural text compression. *IEEE Transactions on Neural Networks*, 7(1):142–146, 1996.

[29] J. 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.

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

[31] J. Schmidhuber, J. Zhao, and N. Schraudolph. Reinforcement learning with self-modifying policies. In S. Thrun and L. Pratt, editors, *Learning to learn*, pages 293–309. Kluwer, 1997.

[32] J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130, 1997.

[33] C. E. Shannon. A mathematical theory of communication (parts I and II). *Bell System Technical Journal*, XXVII:379–423, 1948.

[34] R.J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.

[35] O. Watanabe. *Kolmogorov complexity and computational complexity.* EATCS Monographs on Theoretical Computer Science, Springer, 1992.

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

[37] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

[38] R. J. Williams. Complexity of exact gradient computation algorithms for recurrent neural networks. Technical Report Technical Report NU-CCS-89-27, Boston: Northeastern University, College of Computer Science, 1989.

[39] R. J. Williams and J. Peng. An efficient gradient-based algorithm for online training of recurrent network trajectories. *Neural Computation*, 4:491–501, 1990.

[40] R. J. Williams and D. Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111, 1989.

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

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

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