NMPC: An Algorithm for Online Lossless Data Compression

Thesis (Final Paper) submitted as partial fulfillment of the requirements
towards an MSc degree in Computer Science
The Open University of Israel
Computer Science Division

by

Erez Shermer

Prepared under the supervision of:
Dr. Mireille Avigal
Dr. Dana Shapira

June 2010
Abstract

This work proposes a novel practical and general-purpose lossless compression algorithm named *Neural Markovian Predictive Compression (NMPC)*, based on a novel combination of Bayesian Neural Networks (BNNs) and Hidden Markov Models (HMM). The result is an interesting combination of properties: Linear processing time, constant memory storage performance and great adaptability to parallelism. Though not limited for such uses, when used for online compression (compressing streaming inputs without the latency of collecting blocks) it often produces superior results compared to other algorithms for this purpose. It is also a natural algorithm to be implemented on parallel platforms such as FPGA chips.  

---

1 A paper based on this work was presented in Data Compression Conference 2010, Snowbird, Utah.
Contents

1 Introduction 4
   1.1 Neural Networks 4
   1.2 Hidden Markov Models 8
   1.3 Previous Work 9

2 Description of the Algorithm 11
   2.1 Components 11
   2.2 Flow 14
   2.3 The Neural Markovian Predictive Decompression Algorithm 19
   2.4 Complexity 21

3 Experimental Results 23

4 Conclusions 27
Chapter 1

Introduction

This work presents a lossless compression algorithm named NMPC. The algorithm is based on prediction methods known in the Artificial Intelligence field as Bayesian Neural Networks and Hidden Markov Models. The following subsections present a short introduction to these methods.

1.1 Neural Networks

An artificial Neural Network (NN) may be represented as a weighted directed graph. To maintain some similarity to the biological model, we shall call the vertices in such a graph neurons and the arcs axons. We focus on feedforward networks in which the graph is acyclic and can therefore be seen as a set of ordered layers – the first layer being the neurons with no input axons and the last layer being the neurons with no output axons. Each neuron has a predefined activation function – a derivable function used as the neuron’s final processing of its output. Each neuron in the network can have a unique activation function. The most widely used activation functions in neural networks are linear functions, sigmoids and hyperbolic tangents.
A neural network can be activated as follows. At first, input values are assigned to the neurons of the first layer – thus this layer is often referred to as the input layer. Each neuron calculates the results of its activation function using its own value as a parameter, and then passes the result to all the neurons it is connected to. In turn, these neurons compute the weighted average of their input values, apply their activation function and pass the result to the next layer. The output of neuron $k$ which is passed to the next layer is marked as $o_k$ and thus:

$$o_k = f_k \left( \sum_{i \in \text{Pred}(k)} w_{ik} o_i \right)$$

where $f_k$ is the activation function of neuron $k$, $\text{Pred}(k)$ is a set of the neurons connected to $k$ from the previous layer and $w_{ik}$ is the weight of the axon connecting neuron $i$ to neuron $k$.

The values of the last layer (after computing the weighted average and applying its own activation functions), is the result of the network’s activation, thus this layer is called the output layer. It is easy to see that if we assign the variables $x_1, \ldots, x_I$ to the input layer of a network with $I$ input neurons, the resulting output is actually a vector which is a function $F(x_1, \ldots, x_I)$ made of linear combinations and compositions of activation functions. To add another degree of freedom to this function an extra bias neuron, which is a neuron that always outputs 1 and is connected to all other neurons, is often added.

A sample neural network structure and activation is given in Figure 1. Figure 1a shows a sample network with 3 layers – input, hidden and output. Figure 1b demonstrates a network activation with the values 0.5 and 0.9. The first step of the activation, assignment to the input layer, is in blue. The values of the hidden layer are then evaluated (in green), then finally the output layer (in red).
Figure 1: Top to bottom: (a) Neural network structure; (b) A sample activation; (c) Training
The most useful property about neural networks is training: a process of gradually changing the axon weights so it will converge to a given function. The network is presented with a set of examples – tuples of input vectors and their desirable output vectors called targets. For each such example, the network is activated with the input vector, and the difference between the resulting output and the corresponding target is examined. This difference is called the error vector and often notated as $J$ (shown in red on Figure 1c). A partial derivative of this vector is calculated with respect to each of the network’s axon weights ($\partial J/\partial w_{ik}$) using a process called backpropagation. The process is based on first deriving the error vector with respect to the weights connecting the output nodes and then propagating the derivatives downward. Finally when all the partial derivatives are available, the weights can be re-calibrated to find a minimum point of the error vector using numerical methods such as Newton-Raphson. More details are available in [4].

One special kind of neural network is a Bayesian Neural Network (BNN). This kind of network works the same as described above, but it is trained so that the output neurons represent values of conditional probabilities. Given a set of $n$ events $A_1, A_2, \ldots, A_n$ and their probabilities ($\sum_{1 \leq i \leq n} P(A_i) = 1$), each output neuron $o_i$ represents the probability $P(A_i|X)$ where $X$ is the vector assigned to the network input layer. In this configuration each example presented to the network will be of the form $< X, V >$, $X$ being an input vector of size $I$ and $V$ being a target vector with the same size as the network’s output layer. Each such example is interpreted as “when the input was $X$, the event $A_k$ occurred”, thus for this example the network should optimally give $P(k|X) = 1$ and therefore $V = (0, \cdots, 0, 1, 0, \cdots, 0)$ having 1 only in the $k^{th}$ component.
1.2 Hidden Markov Models

A Markov Model is an automaton in which the transition between states is a random process. Given the states $s_1, \ldots, s_n$, “activating” the model creates a chain of random visits by the conditional probabilities $P(s_i|s_j)$ ($1 \leq i, j \leq n$), also known as a Markov Chain.

A Hidden Markov Model (HMM) is an extension to this idea, in which the model represents a closed unit that is inaccessible to external observers (thus “hidden”). The observer can see only a chain of observations generated by the model using a second random process. For each symbol $\sigma_i$ of some alphabet $\Sigma$ and each state $s_j$ of the model, a value $P(\sigma_i|s_j)$ is defined as the probability of outputting $\sigma_i$ while in the state $s_j$.

Since the HMM outputs one symbol for each visited state during its activation, an external observer who inspects a string $S$ may try to perform various estimations. He can, for example, try to estimate the probability that the HMM visits a specific sequence of states to generate $S$. Given the HMM in Figure 2, the string $aa$ was most likely generated by the sequence $A \rightarrow D \rightarrow B$. He can also try to estimate the probability that a specific HMM is the one that generated $S$. For example, it is easy...
to see that the string \textit{bba} is generated by the sample HMM in a chance of 0.00224
\((= 0.7 \cdot 0.4 \cdot 0.4 \cdot 0.1 \cdot 1 \cdot 0.2)\). It is much more likely that a string like \textit{aab} will be
generated by it, as this string has a probability of \(0.7 \cdot 0.6 \cdot 0.4 \cdot 0.9 \cdot 1 \cdot 0.2 = 0.03024\) to
be generated. This calculation is relatively simple because in this example only one
route can generate strings of length 3. In most cases the string can be generated by
several routes and their corresponding probabilities must be added. We will focus on
this later task, which is accomplished using a method called the \textit{Forward Algorithm}.
This is a dynamic-programming algorithm which is an extension of the brute-force
calculation we used in the last two examples. It incrementally calculates values in a
matrix in which each cell represents the probability that the HMM generated \textit{S} up
to a given index and ended in a certain state. The sum of the last matrix column is
the probability of the HMM generating \textit{S}. The algorithm is fully described in [4].

Additionally, the \textit{Forward-Backward} algorithm allows training the HMM. It allows
the HMM to adapt to “accepting” a given string, thus increasing the probability that
it would really produce it. This is performed in a similar manner to the Forward
Algorithm, using a second “backward” matrix (representing probabilities of the HMM
generating the suffix of \textit{S}) and applying an expectation minimization technique. See

\subsection{1.3 Previous Work}

In the last 15 years a great progress has been made in the theory and practice of
Neural Networks and Hidden Markov Models. Proving to be useful for various tasks,
several attempts have been made to utilize the unique properties of NNs and HMMs
for data compression.
In 1995 Forchhammer & Rissanen [6] suggested an expansion of HMM using a combination with a regular Markov Model and used the new model for compression of binary images. The compression is based on having the model learn the image, then saving its resulting parameters.

In 1997 Booksten, Klein & Raita [2] showed that concordances can efficiently be compressed using Markov models (both hidden and regular). They coded concordances using bit vectors and used a Markov model to represent the movement between clusters of 1s and 0s. A similar idea of cluster prediction was used by the same researchers in 2000 [1] to compress bit images using a Bayesian network (not a neural network).

In 2003 it was suggested by Yang Guowei, Li Zhengxhi & Tu Xuyan [9] that neural networks can be used for generic lossless compression by “folding” a bit stream into \( N \) dimensions and training a neural network to approximate it. A coded form of the resulting network was saved as the output. The idea was new but the resulting compression ratio was only 5.5 bits per character (bpc).

In 2006 Durai & Saro [5] proposed an algorithm for compressing images using neural networks by approximation. The focus there was a transformation proven to greatly improve the convergence time of the network.

The proposed algorithm of this work, Neural Markovian Predictive Compression (NMPC), uses neural networks and HMMs in very different manners than previous works. The neural network here is Bayesian and used for prediction, not for approximation. It is used to filter and pre-order results before sending them to HMMs for delicate inspection. Being adaptive, this unique structure does not require storing the HMMs or BNN data in the output file – it is reconstructed during decompression.
Chapter 2

Description of the Algorithm

This section presents the NMPC algorithm developed in this work. Section 2.1 details the various components used by the algorithm. Section 2.2 explains how these components are used together and presents a pseudo-code for NMPC.

2.1 Components

The NMPC algorithm is composed of a single BNN and $|\Sigma|$ HMMs denoted by $m_1, ..., m_{|\Sigma|}$, where $\Sigma = \{\sigma_1, \sigma_2, ...\}$ is the alphabet. During the execution of NMPC the BNN is gradually trained to answer queries of the type – “given that the last three input characters were $abb$, what is the probability that the next input character will be $a$?”.

We adapt the notation $c[i, j]$ for representing the substring $c_i c_{i+1} \cdots c_j$ of the input string $c$. Given a string $c[k; k + I - 1]$ of length $I$, a single network activation will produce a vector $v = (v_1, ..., v_{|\Sigma|})$, where

$$v_j = P(c_{k+I} = \sigma_j | c[k, k + I - 1])$$

In other words, $v$ is a vector in which component $j$ represents the probability that
the next input character will be $\sigma_j (\in \Sigma)$ given the previous $I$ input characters. Note that this usage of probabilities is only an estimate – the BNN may produce different results in various stages of its learning process.

The HMMs have a similar role with a slightly different configuration: each HMM is responsible for the approximation of only one such conditional probability. Executing the Forward Algorithm on $m_j$ given the string $c[k, k+I-1]$ will produce an estimate for the same probability as $v_j$. (Actually the algorithm uses only a suffix of size $T \leq I$, as experiments show that the HMMs perform best with smaller suffixes.) This estimate is expected to be different than its BNN counterpart for either better or worse, based on the strengths and weaknesses of each method. It can generally be said that while the BNN is good with general data statistics and mathematical connections between data sets, HMMs are good with long-term memory for patterns. The BNN is actually a dynamic global prediction function which does not remember every single character combination. The HMMs, however, take into account the entire previous input stream – thus the two methods complete each other.

Since the BNN produces a probability for each alphabet character, it must have exactly $|\Sigma|$ output units. The size of its input layer is a fixed parameter $I$. In addition it has a single hidden layer in a fixed size $H$. Adjacent layers are fully-connected and therefore the total number of axons in the network is $H (|\Sigma| + I)$. The chosen activation functions for the network are linear for the input and output layers, and $tanh^1$ for the hidden layer.

One last important parameter for the neural network is its input normalization radius. The alphabet letters are represented as integers between 1 and $|\Sigma|$, but the dynamic response of the $tanh$ activation function is roughly in a narrow symmetric

$1tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - 1}{e^x + 1}$
area around $[-\pi/2, \pi/2]$. Therefore a parameter $R$ is used for normalizing the character values to a continuous range $[-R, R]$. Normalization is performed by a simple linear transformation:

$$Norm(\sigma) = 2R \cdot \frac{\sigma}{|\Sigma|} - R$$

The same range is used when presenting the network with target probabilities and when reading its output probabilities. Computational methods for finding “good” values of $R$ are available, but here it was chosen according to empirical experiments.

The HMMs in use are ergodic (the states are fully connected) and have $S$ states each ($S$ is a constant integer). As explained above, each HMM is used as a statistical model for predicting the appearance of a single character of $\Sigma$. Given a parameter $T$ ($T \leq I$) determining the string prefix length passed to the HMMs (similar to $I$ for the BNN), it is natural to represent each HMM using two matrices of order $S \times T$ - one for the transition probabilities and the second for the characters that can be output in each transition.

Each HMM is trained using the Forward-Backward algorithm. Unlike the regular usage of this algorithm, after setting new transition probabilities for moving out of a state, the transition probabilities are modified to ensure that the HMMs stays ergodic – meaning no transition has 0 probability – or for practical matters smaller than some small $\epsilon$. The reason is that probabilities of 0 make it difficult for the HMMs to handle rare combinations of characters. This is performed by counting the number of outgoing arcs $k$ having a transition probability smaller than $\epsilon$, and for each such arc $e_i$ applying a new probability:

$$P_{new}(e_i) = \begin{cases} 
    P(e_i) \cdot (1 - U) & P(e_i) > \epsilon \\
    \frac{U}{k} & P(e_i) \leq \epsilon
\end{cases}$$
where $U$ is a predefined positive constant smaller than 1.

Each HMM $m_j$, $1 \leq j \leq |\Sigma|$, is gradually trained using the Forward-Backward algorithm to estimate the conditional probability $P(c_{k+T} = \sigma_j | c[k, k + T - 1])$. As explained in [4] this can be done in $O(T \cdot S^2)$ time.

### 2.2 Flow

The NMPC algorithm has two main phases: Initial training and actual compression. The first is meant for an initial training in which the first $K$ input characters are written to the output with no modification while the BNN and HMMs are trained.

The second phase is the compression phase. For each prefix of the input stream to be compressed, an intermediate product of this phase is a list $L$, which represents a forecast for the appearance of the next character. For each such prefix $c[i-1+1, i-1]$, the first character of $L$ is the one determined to have the greatest chance of being the next character in the input, $c_i$. Therefore if we look for position $y_i$ of the actual character $c_i$ in $L$, a successful construction of $L$ will usually produce low values of $y_i$.

For instance, in an ASCII coding of text files it was empirically shown that $y_i = 1$ for about 23% of the characters, and $y_i \leq 20$ for about 95%. This means that for 95% of the characters in text files, the next character for each prefix will be among the first 20 items in the list $L$ built for that prefix. (This was measured on text files listed in the experimental results section below.)

The values of list positions $y_i$ are the final output of NMPC’s prediction mechanism and are then coded to the output. In this work Arithmetic Coding [7] was chosen, but any other coding method can be used according to the circumstances. Since the AC algorithm requires a histogram for its input values, NMPC maintains a histogram
of the produced $y_i$ values. Note that the histogram is of the $y_i$ values and not the input values: following the example above, for text files we would get $\text{hist}[1] = 0.23$ and $\sum_{1 \leq j \leq 20} \text{hist}[j] = 0.95$.

The pseudo-code for the streaming NMPC algorithm is presented in Figure 3. NMPC is given an input string $c[1,n]$, and it generates a compressed stream. In addition it is supplied the parameters $S$ (number of states), $R$ (normalization radius), $I$ (input layer size), $T$ (string prefix size for HMM predictions), $K$ (initial buffer size), $B$ (number of iterations of backpropagation), and $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$. Initialization of the BNN network is done by initializing its input layer, sending each neuron a normalized character (according to $R$). In addition, a training vector $v = (-R, -R, \ldots, R, \ldots, -R)$ is constructed so that its $j^{th}$ component is $R$, while all other components are $-R$, given that $c_i$ is the $j^{th}$ character of $\Sigma$, i.e. $c_i = \sigma_j$. This is the same process as described for BNNs in Chapter 1.1, only normalized to $R$.

In order to train the network according to vector $v$ backpropagation is then applied, running $B$ iterations, where at each iteration the network’s output layer is compared to $v$. The Forward-Backward algorithm is then used to train $m_j$ according to the substring $c[i-T+1,i-1]$, of $T$ characters that precede it. If $c_i$ is among the characters of the initialization stage, i.e., it is part of the first $K$ characters of $c$, the character is simply output to the compressed file, and the process continues with the following character of $c[1,n]$ ($i$ is incremented).

Once the training for the first $K$ characters is complete, each additional character goes through the **encoding phase**. In this phase the current knowledge of the BNN and HMMs is used to efficiently encode $c_i$. To do so, the current previous substring of $I$ characters long, i.e. substring $c[i-I+1,i-1]$ of the input string, is normalized and is assigned to the network’s input layer. For simplicity we refer to characters instead
to their normalized values throughout the pseudo-code with the understanding that the BNN is only activated with the normalized values. A list $L$ of probabilities is obtained from the output layer of the network, where each probability is associated to a character of the alphabet $\Sigma$. The list $L$ is then sorted in descending order, so the first character in $L$ is the one predicted by the BNN to have the highest chance of being equal to $c_i$ (note that the network is not yet familiar with $c_i$ at this point). The first $D$ probabilities of $L$ are then re-estimated in order to incorporate the capabilities of the Markov Model. For each character of the first $D$ elements of $L$, if $k$ is the index of that character in $\Sigma$, the Forward Algorithm is applied on $m_k$, in order to compute $P(c_i = \sigma_k|c[i-T+1,i-1])$. The first $D$ elements of $L$ are resorted with these new estimations. The position $y_i$ of $c_i$ in $L$ is then encoded using any static or adaptive coder and the process continues with the following character of $C$. In this case an adaptive Arithmetic Coder was used. The main idea of this phase is illustrated in Figure 4. In this example the string $aadgbb...abbb$ is already processed and the following character should be encoded next. Step 1 shows the list built by the BNN, step 2 shows how it is modified by the HMMs and in step 3 the actual next character is exposed and found to be the second item in $L$.

The encoding phase, activated for $c_i$ for all $K+1 \leq i \leq n$, must always take place before the training phase. This ensures that the encoding of each $c_i$ uses a BNN and HMMs that had no prior knowledge of $c_i$, which is a crucial invariant for allowing the output to be decompressed. The training procedure is repeated after each encoding to keep the BNN and $m_i$’s tuned, and it may be skipped when reaching some bound on the number of characters in order to boost performance. It can also be skipped when prior knowledge determines that statistics are stable, and probabilities don’t change more than some given (tiny) lower bound.
\textbf{NMPC}(c[1, n], out)

1: // Input: a string of \(n\) characters to compress, output: a compressed string
2: // The first \(K\) characters are copied to the output and are used for training
3: \(\text{out}[1, K] \leftarrow c[1, K]\)
4: \textbf{for} \(i \leftarrow I + 1\) \textbf{to} \(n\) \textbf{do}
5: \hspace{1em} \textbf{if} \(i > K\) \textbf{then}
6: \hspace{2em} // Encoding phase
7: \hspace{2em} \(BNN \leftarrow c[i - I, i - 1]\) // Activation, in fact normalized values are used
8: \hspace{2em} \(L \leftarrow BNN\) // Build \(L\) by the network output layer
9: \hspace{2em} \text{sort}(L)
10: \hspace{2em} \textbf{for} \(k \leftarrow 1\) \textbf{to} \(D\) \textbf{do}
11: \hspace{3em} \(z \leftarrow \text{index of } L_k \text{ in } \Sigma\)
12: \hspace{3em} \text{Forward}(m_z, c[i - T, i - 1])
13: \hspace{2em} \textbf{end for}
14: \hspace{2em} \text{sort(first } D \text{ elements of } L\)
15: \hspace{2em} \(y_i \leftarrow \text{index of } c_i \text{ in } L\)
16: \hspace{2em} \text{code}(y_i, out) // Using adaptive coding
17: \hspace{1em} \textbf{end if}
18: // Training phase
19: \hspace{1em} \(j \leftarrow \text{index of } c_i \text{ in } \Sigma \text{ (giving } c_i = \sigma_j\))
20: \hspace{1em} \(v \leftarrow (-R, -R, \ldots, R, \ldots, -R) \text{ / / } j^{th} \text{ component is } R\)
21: \hspace{1em} \text{Backprop}(BNN, c[i - I, i - 1], v) // Train for the \(I\) characters before \(c_i\)
22: \hspace{1em} \text{ForwardBackward}(m_j, c[i - T, i - 1]) // Train for the \(T\) characters before \(c_i\)
23: \textbf{end for}

Figure 3: The NMPC algorithm pseudo-code
The first few elements of L are given to HMMs for "second opinion". In this case the HMM of 'b' and the HMM of 'c' return the highest probabilities.

The real character is exposed. It is 'c', placed second in the list.
2.3 The Neural Markovian Predictive Decompression Algorithm

The NMPC-decompression algorithm is symmetric to the NMPC-compression algorithm, and is fully listed in Figure 5. Since compression is performed on a single character at a time, decompression can be done incrementally by working on the compressed stream. The BNN and HMMs are trained on the same initial $K$ characters, which were written explicitly to the output stream during the compression process. All parameters are also supplied to the decompressor; in particular, the value $K$ is known during decompression, so that it concludes that no decoding is needed for the first $K$ characters. Once reading the first $K$ characters of the input stream, it is directly transferred to the decompressed stream, the same network is initialized and activated and the same HMMs are trained. After the first $K$ characters are dealt with, the current prefix of the stream corresponds to the encoding of some list index $x$ in the soon-to-be-constructed list $L$. Once the list is constructed for the first character after the $K$-sized prefix, it is identical to the list $L$ built for the same index in the compression algorithm. It can be proved by induction on the number of iterations that this invariant holds – the list $L$ built after terminating the $i^{th}$ iteration in the compression method is identical to the list built after terminating the $i^{th}$ iteration in the decompression method. Therefore decompression requires outputting the character in position $x$ of $L$, and the process continues with the remaining part of the input stream until the end of the stream is reached.
NMPCDecompress(s[1,n], out)

1: // Input: a compressed string s of length n, output: a decompressed string
2: // The first K characters are copied to the output and are used for training
3: out[1,K] ← s[1,K]
4: // i is used as the output index
5: for i ← I + 1 to K do
6: // Training phase for the initial K characters
7: j ← index of s_i in Σ (giving s_i = σ_j)
8: v ← (−R, −R, . . . , R, . . . , −R) // jth component is R
9: Backprop(BNN, s[i − I, i − 1], v) // Train for the I characters before s[i]
10: ForwardBackward(m_j, s[i − T, i − 1]) // Train for the T characters before s[i]
11: end for
12: while (input stream still has characters to decode) do
13: decode(x, c) // Decode one index x using the adaptive decoder
14: BNN ← out[i − I, i − 1] // Activation, in fact normalized values are used
15: L ← BNN // Build L by the network output layer
16: sort(L)
17: for k ← 1 to D do
18: z ← index of L_k in Σ
19: Forward(m_z, out[i − T, i − 1])
20: end for
21: sort(first D elements of L)
22: out[i] ← L_x // The character at position x
23: i ← i + 1
24: end while

Figure 5: The NMPC Decompression algorithm pseudo-code
2.4 Complexity

NMPC can naturally be adapted to parallel execution, as its computation components such as BNN activation, BNN backpropagation, HMM Forward, and HMM Forward-Backward algorithms work on many independent units. The operations can be performed simultaneously on individual neurons of the BNN network and/or the HMM states. Platforms that allow a big number of independent simple execution units, such as FPGA chips, can exploit this property of NMPC to achieve great performance and simple design. Being this a strong property of the NMPC algorithm, the analysis of processing time complexity is given for both the case of serial machines and the case of full parallelism. In order to accomplish optimal parallel computation, we assume that $\max(H \cdot (|\Sigma| + I), D \cdot S)$ parallel execution units are available.

The complexity is analyzed in terms of the following constants, all previously described in their appropriate context: $I$ (the size of the network’s input layer), $H$ (the size of the network’s hidden layer), $\Sigma$ (size of the alphabet), $T$ (size of the string suffix for HMM processing), $S$ (number of HMM states), $B$ (number of backpropagation iterations), $D$ (number of list elements for HMM ”second opinion”) and $n$ (number of characters in the original string).

Populating the network’s input layer of size $I$ takes $O(I)$ for $I$ independent neurons in serial execution, or $O(1)$ parallel time. Since the BNN has exactly $|\Sigma|$ output units, attaining the information from the output layer takes $O(|\Sigma|)$ processing time with serial computation or $O(1)$ in parallel. Backpropagation with $B$ iterations takes $O(B \cdot H (|\Sigma| + I))$ in serial computation or $O(B)$ in parallel, where $H \cdot (|\Sigma| + I)$ is the number of axons. The Forward-Backward algorithm’s serial processing time is $O(T \cdot S^2)$ for HMM with $S$ states and a buffer of size $T$, or $O(TS)$ in parallel. Sorting list $L$ of $\Sigma$ items takes $O(|\Sigma| \log |\Sigma|)$ for serial computation and only $O(\log |\Sigma|)$ for
parallel computation. The Forward Algorithm is only applied on the first $D$ elements and therefore takes $O(D \cdot (T \cdot S^2))$ in serial processing time or just $O(TS)$ in parallel. Resorting only $D$ elements, and performing one step of Arithmetic Coding is drowned out by the bigger time consuming sort of $\Sigma$ elements. For the total processing performance computation, by referring to the most time consuming methods, we sum up Backpropagation, HMM Forward, HMM Forward-Backward, and sorting algorithms for a total of $O(n \cdot (B \cdot H(|\Sigma| + I) + D \cdot T \cdot S^2 + |\Sigma| \cdot \log |\Sigma|)$ serial time, and only $O(n \cdot (B + TS + \log |\Sigma|))$ parallel time. NMPC is, therefore, linear in the size of the string to be compressed, both parallel and serial, but with smaller hidden constants for the parallel case. Although the supplied parameters have no effect on processing time complexity the actual running time is affected.
Chapter 3

Experimental Results

NMPC was tested as an online compression algorithm on a big number of input files and compared to three popular algorithms. The first is Arithmetic Coding (AC), which adaptively codes the input into a stream in which each character may get a non-integer number of bits [7]. The second is Lempel Ziv Welch (LZW), which uses a dynamically built dictionary and codes indexes of dictionary entries that replace symbol combinations [8]. The last is Burrows-Wheeler Transform (BWT) which uses a sophisticated sorting of string permutations, and together with the Bring to Front transformation (BWT) and AC often produces impressive results [3].

LZW is well-suited for online compression, as it does not require the input stream to be divided into blocks. Using a fixed size dictionary, LZW can be implemented in linear time. However, unlike NMPC, it does not significantly utilize parallelism. LZW was tested with dictionary sizes of 1024 entries (LZW1024) and 2048 (LZW2048) in order to simulate a similar memory consumption to the tested NMPC instance.

BWT (combined with MTF and AC) is not ideal for online compression; it must operate on whole blocks of characters so it is not a natural candidate for low-latency uses. However for some practical uses of online compression small blocks of 512 or
1024 bytes can be a reasonable compromise.

Arithmetic Coding is a good choice for online compression and is used as a component with BWT and in the version of NMPC tested here. Being an entropy encoder, for some inputs it is the most efficient compression method by itself.

Parameters for NMPC were chosen to produce a good average ratio for text files; different parameters for different files will produce different results.

The 6 algorithms – NMPC, LZW1024, LZW2048, BWT512, BWT1024, AC – produced the results shown in Figure 6. The best result in each row is shown in bold.

The results clearly demonstrate NMPC’s strengths and weaknesses. NMPC is good with text files and gets the best results for more files than any other algorithm tested. For the cases it does not achieve the best compression performance, its compression results tend to be close to those of the best method. It is not better than LZW when it comes to compressing source code and some data files, as the repetitive nature of these files is very suitable for dictionary-based compression.

The above results were measured using a single set of parameters for the NMPC algorithm. The parameters were chosen to achieve good average results for text files and are not optimal for each file separately. For each given file, there is usually a set of parameters for which NMPC achieves a better compression ratio than measured above. Therefore it would be natural to improve NMPC by adding a parameter estimation phase for each file (based only on the first $K$ characters so the decompression algorithm can find the same parameters). To demonstrate this point, Figure 7 shows the result of NMPC compressing the file alice.txt (from Canterbury Corpus) after changing the value of a single parameter at a time, keeping all the other parameters unchanged. The default value of each parameter is shown in bold.
<table>
<thead>
<tr>
<th>Original</th>
<th>NMPC</th>
<th>LZW1024</th>
<th>LZW2048</th>
<th>BWT512</th>
<th>BWT1024</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wikipedia Compression</td>
<td>8,776</td>
<td><strong>4,809</strong></td>
<td>5,539</td>
<td>4,952</td>
<td>6,570</td>
<td>5,567</td>
</tr>
<tr>
<td>Wikipedia Israel</td>
<td>56,783</td>
<td><strong>33,767</strong></td>
<td>38,308</td>
<td>35,167</td>
<td>44,083</td>
<td>37,525</td>
</tr>
<tr>
<td>Alice (Guttenberg)</td>
<td>147,800</td>
<td><strong>82,368</strong></td>
<td>96,400</td>
<td>87,443</td>
<td>113,545</td>
<td>96,789</td>
</tr>
<tr>
<td>Dracula (Guttenberg)</td>
<td>874,665</td>
<td>490,189</td>
<td>569,588</td>
<td>320,338</td>
<td>671,354</td>
<td>575,729</td>
</tr>
<tr>
<td>Far From (Guttenberg)</td>
<td>815,934</td>
<td><strong>455,619</strong></td>
<td>543,613</td>
<td>496,432</td>
<td>634,359</td>
<td>546,037</td>
</tr>
<tr>
<td><strong>English Text</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Various</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>numbers.txt</td>
<td>20,265</td>
<td><strong>6,870</strong></td>
<td>10,994</td>
<td>10,452</td>
<td>14,958</td>
<td>13,559</td>
</tr>
<tr>
<td>NeuralApproximator.cpp</td>
<td>33,430</td>
<td>20,667</td>
<td>20,998</td>
<td><strong>18,565</strong></td>
<td>24,472</td>
<td>20,208</td>
</tr>
<tr>
<td>Bach.mp3</td>
<td>364,974</td>
<td>364,006</td>
<td>451,260</td>
<td>491,184</td>
<td>376,906</td>
<td>374,002</td>
</tr>
<tr>
<td><strong>Canterbury Corpus</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alic29.txt</td>
<td>152,089</td>
<td><strong>84,951</strong></td>
<td>97,934</td>
<td>88,863</td>
<td>116,839</td>
<td>100,017</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125,179</td>
<td><strong>74,333</strong></td>
<td>84,061</td>
<td>76,803</td>
<td>96,588</td>
<td>82,493</td>
</tr>
<tr>
<td>cp.html</td>
<td>24,603</td>
<td>16,230</td>
<td>16,976</td>
<td><strong>14,980</strong></td>
<td>19,063</td>
<td>15,961</td>
</tr>
<tr>
<td>fields.c</td>
<td>11,150</td>
<td>7,158</td>
<td>6,274</td>
<td><strong>5,843</strong></td>
<td>7,775</td>
<td>6,219</td>
</tr>
<tr>
<td>grammar.lisp</td>
<td>3,721</td>
<td>2,405</td>
<td>1,965</td>
<td><strong>1,944</strong></td>
<td>2,495</td>
<td>2,049</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1,029,744</td>
<td>561,340</td>
<td>267,551</td>
<td>279,270</td>
<td>480,972</td>
<td><strong>243,231</strong></td>
</tr>
<tr>
<td>lcet10.txt</td>
<td>426,754</td>
<td><strong>244,835</strong></td>
<td>275,946</td>
<td>250,995</td>
<td>329,046</td>
<td>278,253</td>
</tr>
<tr>
<td>plrabdn12.txt</td>
<td>481,861</td>
<td><strong>265,996</strong></td>
<td>318,334</td>
<td>290,937</td>
<td>375,671</td>
<td>325,578</td>
</tr>
<tr>
<td>pt5</td>
<td>513,216</td>
<td>97,680</td>
<td>75,038</td>
<td><strong>72,050</strong></td>
<td>253,243</td>
<td>173,094</td>
</tr>
<tr>
<td>sum</td>
<td>38,240</td>
<td>25,950</td>
<td>21,739</td>
<td><strong>21,355</strong></td>
<td>26,231</td>
<td>22,021</td>
</tr>
<tr>
<td>xargs.1</td>
<td>4,227</td>
<td>2,682</td>
<td>2,806</td>
<td><strong>2,470</strong></td>
<td>3,236</td>
<td>2,702</td>
</tr>
<tr>
<td><strong>Large Canterbury</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bible.txt</td>
<td>4,047,392</td>
<td><strong>2,083,747</strong></td>
<td>2,381,805</td>
<td>2,134,541</td>
<td>2,935,193</td>
<td>2,423,235</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2,473,400</td>
<td>1,584,820</td>
<td>1,706,416</td>
<td>1,594,668</td>
<td>1,941,345</td>
<td>1,697,589</td>
</tr>
<tr>
<td>E.coli</td>
<td>4,638,690</td>
<td><strong>1,158,567</strong></td>
<td>1,576,274</td>
<td>1,488,731</td>
<td>2,650,363</td>
<td>2,051,540</td>
</tr>
<tr>
<td><strong>Large Calgary</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trans</td>
<td>93,695</td>
<td>65,170</td>
<td>59,909</td>
<td><strong>55,178</strong></td>
<td>69,955</td>
<td>58,654</td>
</tr>
<tr>
<td>progp</td>
<td>49,379</td>
<td>30,706</td>
<td>27,584</td>
<td><strong>25,201</strong></td>
<td>35,455</td>
<td>28,178</td>
</tr>
<tr>
<td>progl</td>
<td>71,646</td>
<td>41,592</td>
<td>39,840</td>
<td><strong>35,125</strong></td>
<td>49,940</td>
<td>40,754</td>
</tr>
<tr>
<td>proge</td>
<td>39,611</td>
<td>25,899</td>
<td>25,896</td>
<td><strong>23,930</strong></td>
<td>29,835</td>
<td>25,222</td>
</tr>
<tr>
<td>pic</td>
<td>513,216</td>
<td>97,680</td>
<td>75,038</td>
<td><strong>72,050</strong></td>
<td>253,243</td>
<td>173,094</td>
</tr>
<tr>
<td>paper1 (Latex code)</td>
<td>53,161</td>
<td><strong>32,751</strong></td>
<td>35,889</td>
<td>32,932</td>
<td>40,872</td>
<td>34,568</td>
</tr>
<tr>
<td>paper2 (Latex code)</td>
<td>82,199</td>
<td><strong>46,707</strong></td>
<td>54,148</td>
<td>49,189</td>
<td>63,090</td>
<td>53,418</td>
</tr>
<tr>
<td>paper3 (Latex code)</td>
<td>46,526</td>
<td><strong>26,725</strong></td>
<td>31,359</td>
<td>28,868</td>
<td>36,133</td>
<td>30,777</td>
</tr>
<tr>
<td>paper4 (Latex code)</td>
<td>13,286</td>
<td><strong>7,765</strong></td>
<td>9,009</td>
<td>8,318</td>
<td>10,197</td>
<td>8,639</td>
</tr>
<tr>
<td>paper5 (Latex code)</td>
<td>11,954</td>
<td><strong>7,677</strong></td>
<td>8,149</td>
<td><strong>7,635</strong></td>
<td>9,128</td>
<td>7,731</td>
</tr>
<tr>
<td>paper6 (Latex code)</td>
<td>38,105</td>
<td><strong>24,763</strong></td>
<td>25,244</td>
<td><strong>23,289</strong></td>
<td>28,880</td>
<td>24,305</td>
</tr>
<tr>
<td>obj2</td>
<td>246,814</td>
<td>205,755</td>
<td>154,990</td>
<td>148,029</td>
<td>179,991</td>
<td><strong>147,977</strong></td>
</tr>
<tr>
<td>obj1</td>
<td>21,504</td>
<td>17,132</td>
<td>14,661</td>
<td><strong>14,638</strong></td>
<td>16,499</td>
<td><strong>14,311</strong></td>
</tr>
<tr>
<td>news (Newsgroup dump)</td>
<td>377,109</td>
<td><strong>243,739</strong></td>
<td>280,603</td>
<td>261,633</td>
<td>302,780</td>
<td>263,802</td>
</tr>
<tr>
<td>geo</td>
<td>102,400</td>
<td><strong>72,269</strong></td>
<td>83,458</td>
<td>83,517</td>
<td>89,098</td>
<td>81,925</td>
</tr>
<tr>
<td>book2</td>
<td>610,856</td>
<td><strong>355,001</strong></td>
<td>405,725</td>
<td>371,265</td>
<td>466,435</td>
<td>394,235</td>
</tr>
<tr>
<td>book1</td>
<td>768,771</td>
<td><strong>430,460</strong></td>
<td>523,950</td>
<td>479,231</td>
<td>601,463</td>
<td>517,512</td>
</tr>
<tr>
<td>bib (Bibliography list)</td>
<td>111,261</td>
<td><strong>68,966</strong></td>
<td>79,539</td>
<td>70,923</td>
<td>89,110</td>
<td>74,622</td>
</tr>
</tbody>
</table>

Figure 6: Experimental Results
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I = 2</td>
<td>I = 3</td>
<td>I = 4</td>
<td>I = 5</td>
<td>I = 6</td>
<td>I = 7</td>
<td></td>
</tr>
<tr>
<td>85,107</td>
<td><strong>84,951</strong></td>
<td>84,902</td>
<td>84,876</td>
<td>84,852</td>
<td>84,856</td>
<td></td>
</tr>
<tr>
<td>H = 2</td>
<td>H = 3</td>
<td>H = 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>86,438</td>
<td><strong>84,951</strong></td>
<td>86,617</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D = 11</td>
<td>D = 12</td>
<td>D = 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85,544</td>
<td><strong>84,951</strong></td>
<td>85,110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S = 2</td>
<td>S = 3</td>
<td>S = 4</td>
<td>S = 5</td>
<td>S = 6</td>
<td>S = 7</td>
<td>S = 8</td>
</tr>
<tr>
<td>84,791</td>
<td>84,786</td>
<td><strong>84,951</strong></td>
<td>84,585</td>
<td>84,216</td>
<td>84,216</td>
<td>84,440</td>
</tr>
<tr>
<td>T = 1</td>
<td>T = 2</td>
<td>T = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87,348</td>
<td><strong>84,951</strong></td>
<td>88,692</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: Influence of various parameters on compressing alice29.txt
Chapter 4

Conclusions

The algorithm developed in this work, Neural Markovian Predictive Compression or NMPC, is a good practical candidate for lossless online compression of various data types. Empirical experiments show that it performs best when the input includes predictable statistical patterns that can be learned by the BNN and HMMs. This includes text files and various textual data lists. It is possible that similar results may be achieved for non-textual data by using different alphabets (e.g. a 10-bit alphabet for data that has “10-bit logic”). NMPC’s advantages become extremely important in parallel environments, since it allows significant-to-optimal utilization of computation units. NMPC also serves as a platform for many possible customizations and it may become a convenient base for expansion. Given its encouraging initial results, using the same platform with different prediction methods may be a promising new direction for similar algorithms.
Bibliography


# List of Figures

1. Top to bottom: (a) Neural network structure; (b) A sample activation; (c) Training .......................................................... 6
2. A sample HMM with four states and an alphabet \{a, b\} ............... 8
3. The NMPC algorithm pseudo-code ........................................... 17
4. Summary of the NMPC compression stage ................................... 18
5. The NMPC Decompression algorithm pseudo-code ....................... 20
6. Experimental Results .......................................................... 25
7. Influence of various parameters on compressing alice29.txt ........... 26
.lossless data compression) algorithm called lossless data compression, which is called Neural Markovian Predictive Compression, or NMPC. The algorithm is called NMPC, which is an acronym for Neural Markovian Predictive Compression, and it is based on the combination of a Bayesian neural network and Hidden Markov Models (HMMs). The unique use and efficiency of the neural network in the HMMs provide NMPC with linear runtime for many important properties, such as constant and flexible work loads when used for data compression. NMPC's simplicity in the compression of blocks (without the intervention of other algorithms) makes it the most efficient among other compression algorithms and other algorithms, such as FPGA.

The algorithm described in this article was presented in the Data Compression Conference 2010, Snowbird, Utah.

1. FPGA.
תוכן העניינים

مبוא .......................... 1

1.1 רשתות ניורונים .......................... 1.1

8 Hidden Markov Models .......................... 1.2

9 עבירות קודמות .......................... 1.3

11 תואר האלגוריתם .......................... 2

11 רכיבי האלגוריתם .......................... 2.1

14 פעולת האלגוריתם .......................... 2.2

19 אלגוריתם הפרישה .......................... 2.3

21 סיבוכיות .......................... 2.4

23 תוצאות .......................... 3

27 מסקנות .......................... 4
ﻌﺒﻮدت תזה לתואר שני במדעי המחשב:

NMPC - אלגוריתם
لدיהיסת midwayמקוון לﻵ
הפסדים

ארז שורר
ת"ת 061246385

منتניה:
דייר מרים אבגרל
דייר דנה שפריר

יוני 2010