The Open University of Israel

Department of Mathematics and Computer Science

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]

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

Contents

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:

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

where f_k is the activation function of neuron k, *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 $x_1, \cdots, x_I$ to the input layer of a network with I input neurons, the resulting output is actually a vector which is a function $F(x_1, \cdots, x_I)$ 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).

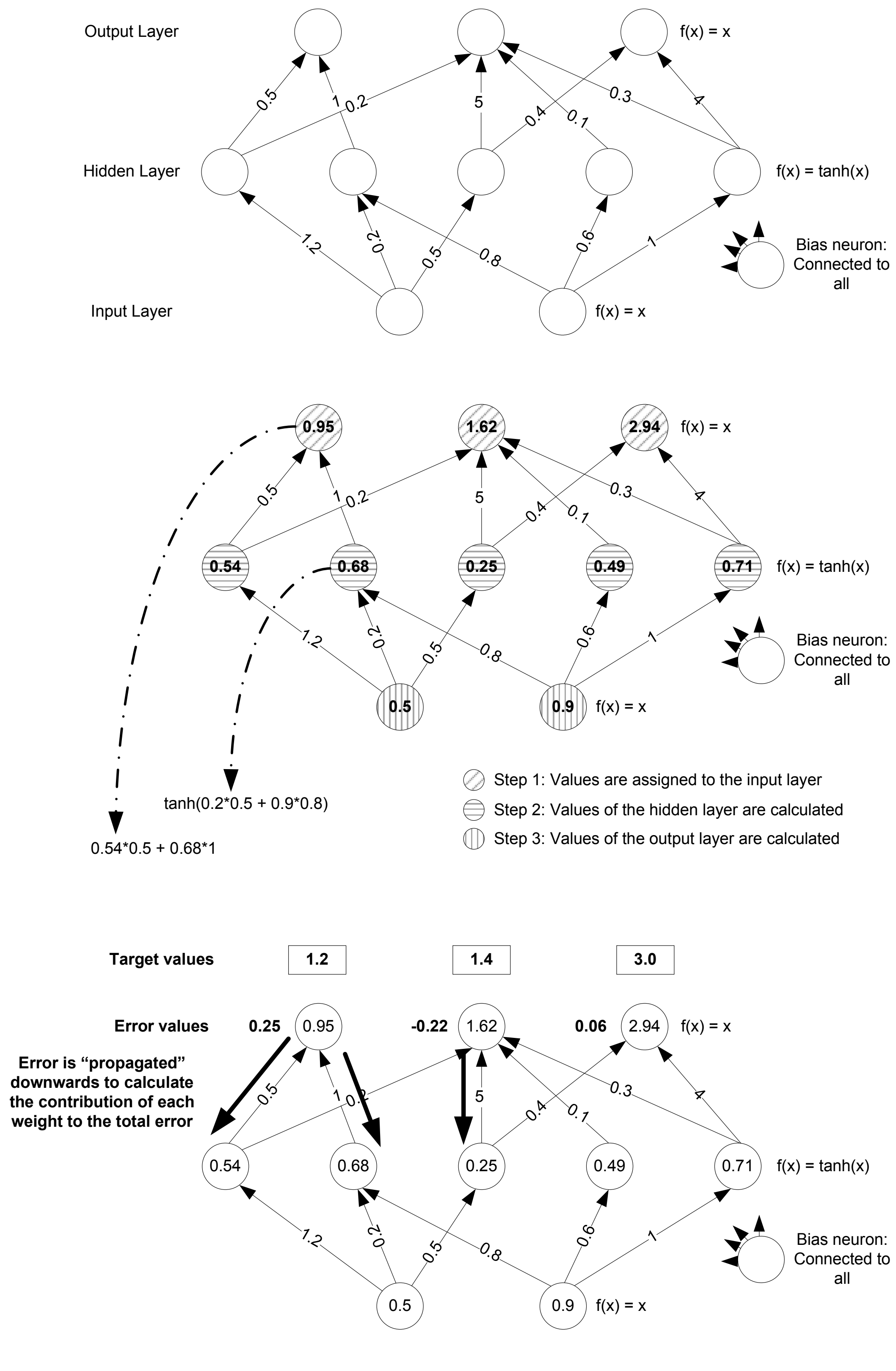
Output Layer
f(x) = x
Hidden Layer
f(x) = tanh(x)
Input Layer
f(x) = x
0.5
1
0.2
5
0.4
0.1
0.3
4
1.2
0.2
0.5
0.8
0.6
1
Bias neuron: Connected to all
0.95
1.62
2.94
0.54
0.68
0.25
0.49
0.71
0.5
0.9
tanh(0.2*0.5 + 0.9*0.8)
0.54*0.5 + 0.68*1
Step 1: Values are assigned to the input layer
Step 2: Values of the hidden layer are calculated
Step 3: Values of the output layer are calculated
Target values
1.2
1.4
3.0
Error values
0.25
-0.22
0.06
Error is "propagated" downwards to calculate the contribution of each weight to the total error

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 $A_1, A_2, ..., A_n$ and their probabilities ($\Sigma_{1 \leq i \leq n} P(A_i) = 1$), 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.

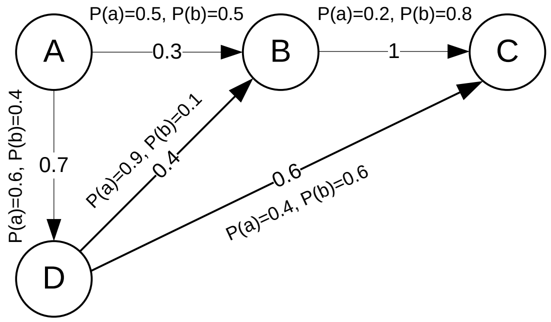

Figure 2: A sample HMM with four states and an alphabet $\{a, b\}$

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 σ_i of some alphabet Σ and each state s_j of the model, a value $P(\sigma_i|s_j)$ is defined as the probability of outputting σ_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 *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 *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 *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 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 S. The algorithm is fully described in [4].

Additionally, the *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 S) and applying an expectation minimization technique. See [4] for more details.

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 researches 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 $v = \left(v_1, ..., v_{|\Sigma|}\right)$, 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 σ_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 $H\left(|\Sigma| + I\right)$. 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

[1] $tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^{2x}-1}{e^{2x}+1}$

area around $[-\frac{\pi}{2}, \frac{\pi}{2}]$. 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 Σ. 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 ϵ. 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 ϵ, 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 $c[i-I+1, i-1]$, 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 $hist[1] = 0.23$ and $\sum_{1 \leq j \leq 20} hist[j] = 0.95$.

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 $\Sigma = \{\sigma_1, \sigma_2, ...\}$. 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 Σ, 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 Σ. 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 Σ, 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...abbbb* 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.

```
$NMPC(c[1,n], out)$
// Input: a string of $n$ characters to compress, output: a compressed string
// The first $K$ characters are copied to the output and are used for training
$out[1,K] \leftarrow c[1,K]$
for $i \leftarrow I+1$ to $n$ do
    if $i > K$ then
        // Encoding phase
        $BNN \leftarrow c[i-I, i-1]$ // Activation, in fact normalized values are used
        $L \leftarrow BNN$ // Build L by the network output layer
        sort($L$)
        for $k \leftarrow 1$ to $D$ do
            $z \leftarrow$ index of $L_k$ in $\Sigma$
            Forward($m_z, c[i-T, i-1]$)
        end for
        sort(first $D$ elements of $L$)
        $y_i \leftarrow$ index of $c_i$ in $L$
        code($y_i, out$) // Using adaptive coding
    end if
    // Training phase
    $j \leftarrow$ index of $c_i$ in $\Sigma$ (giving $c_i = \sigma_j$)
    $v \longleftarrow (-R, -R, \ldots, R, \ldots, -R)$ // $j^{th}$ component is $R$
    Backprop($BNN, c[i-I, i-1], v$) // Train for the $I$ characters before $c_i$
    ForwardBackward($m_j, c[i-T, i-1]$) // Train for the $T$ characters before $c_i$
end for
```

Figure 3: The NMPC algorithm pseudo-code

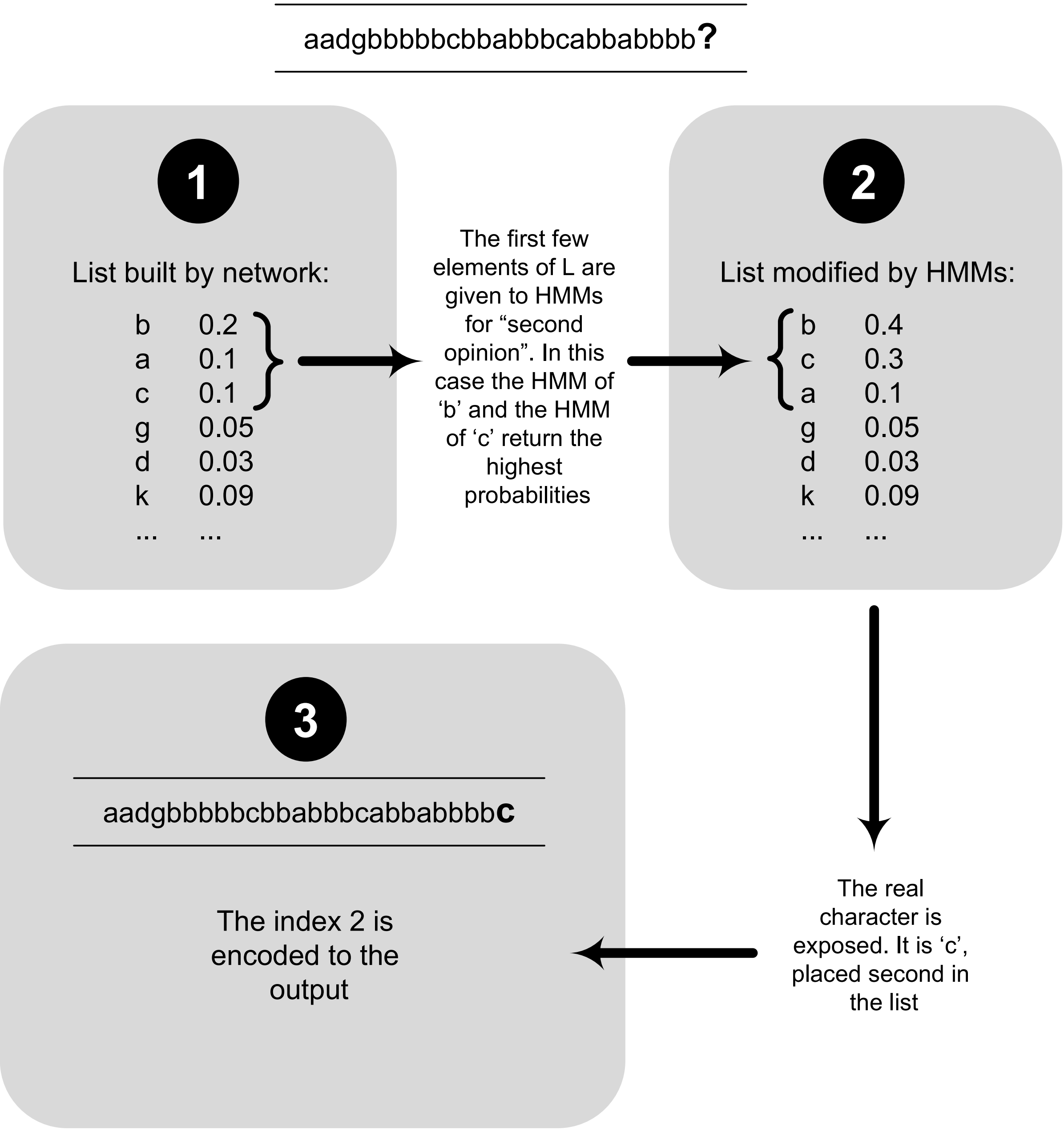
aadgbbbbbcbbabbbcabbabbbb?
1
List built by network:
b 0.2
a 0.1
c 0.1
g 0.05
d 0.03
k 0.09
... ...
The first few elements of L are given to HMMs for “second opinion”. In this case the HMM of ‘b’ and the HMM of ‘c’ return the highest probabilities
2
List modified by HMMs:
b 0.4
c 0.3
a 0.1
g 0.05
d 0.03
k 0.09
... ...
The real character is exposed. It is ‘c’, placed second in the list
3
aadgbbbbbcbbabbbcabbabbbbc
The index 2 is encoded to the output

Figure 4: Summary of the NMPC compression stage

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.

$NMPCDecompress(s[1,n], out)$

```
// Input: a compressed string $s$ of length $n$, output: a decompressed string
// The first $K$ characters are copied to the output and are used for training
$out[1,K] \leftarrow s[1,K]$
// $i$ is used as the output index
for $i \leftarrow I+1$ to $K$ do
    // Training phase for the initial $K$ characters
    $j \leftarrow$ index of $s_i$ in $\Sigma$ (giving $s_i = \sigma_j$)
    $v \longleftarrow (-R, -R, \ldots, R, \ldots, -R)$ // $j^{th}$ component is $R$
    Backprop($BNN, s[i-I, i-1], v$) // Train for the I characters before $s[i]$
    ForwardBackward($m_j, s[i-T, i-1]$) // Train for the T characters before $s[i]$
end for
while (input stream still has characters to decode) do
    decode(x, c) // Decode one index $x$ using the adaptive decoder
    $BNN \leftarrow out[i-I, i-1]$ // Activation, in fact normalized values are used
    $L \leftarrow BNN$ // Build L by the network output layer
    sort($L$)
    for $k \leftarrow 1$ to $D$ do
        $z \leftarrow$ index of $L_k$ in $\Sigma$
        Forward($m_z, out[i-T, i-1]$)
    end for
    sort(first $D$ elements of $L$)
    $out[i] \leftarrow L_x$ // The character at position $x$
    $i \leftarrow i+1$
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), Σ (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 $O\left(B \cdot H\left(|\Sigma| + I\right)\right)$ 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 Σ 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 Σ 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.

	Original	NMPC	LZW1024	LZW2048	BWT512	BWT1024	AC
English Text							
Wikipedia Compression	8,776	**4,809**	5,539	4,952	6,570	5,567	5,175
Wikipedia Israel	56,783	**33,767**	38,308	35,167	44,083	37,525	33,967
Alice (Guttenberg)	147,800	**82,368**	96,400	87,443	113,545	96,789	85,466
Dracula (Guttenberg)	874,665	496,189	569,588	520,338	671,354	575,729	**491,805**
Far From (Guttenberg)	815,934	**455,619**	543,613	496,432	634,359	546,037	466,242
Various							
numbers.txt	20,265	**8,670**	10,994	10,452	14,958	13,559	9,329
NeuralApproximator.cpp	33,430	20,667	20,998	**18,565**	24,472	20,208	22,772
Bach.mp3	364,974	364,006	451,260	491,184	376,906	374,002	**363,683**
Canterbury Corpus							
alice29.txt	152,089	**84,951**	97,934	88,863	116,839	100,017	87,335
asyoulik.txt	125,179	**74,333**	84,061	76,803	96,588	82,493	75,704
cp.html	24,603	16,230	16,976	**14,980**	19,063	15,961	16,373
fields.c	11,150	7,158	6,274	**5,843**	7,775	6,219	7,281
grammer.lsp	3,721	2,405	1,965	**1,944**	2,495	2,049	2,370
kennedy.xls	1,029,744	564,340	267,551	279,270	480,972	**243,231**	454,966
lcet10.txt	426,754	**244,835**	275,946	250,995	329,046	278,253	249,277
plrabdn12.txt	481,861	**265,996**	318,334	290,937	375,671	325,578	273,832
ptt5	513,216	97,680	75,038	**72,050**	253,243	173,094	76,744
sum	38,240	25,950	21,739	**21,355**	26,231	22,021	26,486
xargs.1	4,227	2,682	2,806	**2,470**	3,236	2,702	2,799
Large Canterbury							
bible.txt	4,047,392	**2,083,747**	2,381,805	2,134,541	2,935,193	2,423,235	2,197,739
world192.txt	2,473,400	1,584,820	1,706,416	1,594,668	1,941,354	1,697,589	**1,545,502**
E.coli	4,638,690	**1,158,567**	1,576,274	1,488,731	2,650,363	2,051,540	1,168,765
Large Calgary							
trans	93,695	65,170	59,909	**55,178**	69,955	58,654	65,300
progp	49,379	30,706	27,584	**25,201**	35,455	28,178	30,532
progl	71,646	41,592	39,840	**35,125**	49,940	40,754	43,142
progc	39,611	25,899	25,896	**23,930**	29,835	25,222	26,161
pic	513,216	97,680	75,038	**72,050**	253,243	173,094	76,744
paper1 (Latex code)	53,161	**32,751**	35,889	32,932	40,872	34,568	33,539
paper2 (Latex code)	82,199	**46,707**	54,148	49,189	63,090	53,418	47,632
paper3 (Latex code)	46,526	**26,725**	31,359	28,868	36,133	30,777	27,484
paper4 (Latex code)	13,286	**7,765**	9,009	8,318	10,197	8,639	8,108
paper5 (Latex code)	11,954	7,677	8,149	**7,635**	9,128	7,731	7,694
paper6 (Latex code)	38,105	24,763	25,244	**23,289**	28,880	24,305	24,340
obj2	246,814	205,755	154,990	148,029	179,991	**147,977**	192,114
obj1	21,504	17,132	14,661	14,638	16,499	**14,317**	16,704
news (Newsgroup dump)	377,109	**243,739**	280,603	261,633	302,780	263,802	245,630
geo	102,400	**72,269**	83,458	83,517	89,098	81,925	72,642
book2	610,856	**355,001**	405,725	371,265	466,435	394,235	366,288
book1	768,771	**430,460**	523,950	479,231	601,463	517,512	436,199
bib (Bibliography list)	111,261	**68,966**	79,539	70,923	89,110	74,622	72,723

Figure 6: Experimental Results

I = 2	**I = 3**	I = 4	I = 5	I = 6	I = 7	
85,107	**84,951**	84,902	84,876	84,852	84,856	
H = 2	**H = 3**	H = 4				
86,438	**84,951**	86,617				
D = 11	**D = 12**	D = 13				
85,544	**84,951**	85,110				
S = 2	S = 3	**S = 4**	S = 5	S = 6	S = 7	S = 8
84,791	84,786	**84,951**	84,585	84,216	84,216	84,440
T = 1	**T = 2**	T = 3				
87,348	**84,951**	88,692				

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

[1] T. T. K. A. Bookstein and T. Raita. Simple bayesian model for bitmap compression. *Inf. Retr.*, 1(4):315–328, 2000.

[2] A. Bookstein, S. T. Klein, and T. Raita. Modeling word occurrences for the compression of concordances. *ACM Trans. Inf. Syst.*, 15(3):254–290, 1997.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[4] R. Duda, P. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[5] A. S. Durai and A. E. Saro. Image compression with back-propagation neural network using cumulative distribution function. *Proceedings of the World Academy of Science, Engineering and Technology*, 17(1307-6884):1899–1902, 2006.

[6] S. Forchhammer and J. Rissanen. Coding with partially hidden markov models. In *Data Compression Conference*, pages 92–101, 1995.

[7] N. M. R. I. H. Witten and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

[8] T. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[9] G. Yang, Z. Li, and X. Tu. Lossless data compression methods based on neural networks. *Proceedings of International Conference on Communication Technology*, 2(9-11):1899–1902, 2003.

List of Figures

תקציר

בעבודה זו פותח אלגוריתם מקורי וכללי לדחיסת מידע ללא הפסדים (lossless data compression). האלגוריתם מכונה NMPC, קיצור של Neural Markovian Predictive Compression, והוא מבוסס על שילוב של רשת ניורונים בייסיאנית ו-Hidden Markov Models.

אופן השימוש הייחודי ברשת הניורונים וה-HMM מקנה ל-NMPC כמה תכונות חשובות: זמן ריצה לינארי, סיבוכיות מקום קבועה וגמישות רבה לעבודה מקבילית. כאשר הוא משמש לדחיסה מקוונת (דחיסת מידע זורם ללא השיהוי של איסוף בלוקים) הוא לרוב משיג תוצאות דחיסה טובות יותר ביחס לאלגוריתמים מתחרים. זהו גם אלגוריתם טבעי מאוד למימוש בסביבות מקביליות כמו שבבי FPGA.[1]

[1] המאמר המבוסס על עבודה זו הוצג ב-Data Compression Conference 2010, Snowbird, Utah.

תוכן העניינים

האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי המחשב

עבודת תזה לתואר שני במדעי המחשב:

NMPC - אלגוריתם לדחיסת מידע מקוונת ללא הפסדים

ארז שרמר

ת״ז 061246385

מנחות:

ד״ר מיריי אביגל

ד״ר דנה שפירא

יוני 2010