TEXT FILE COMPRESSION BY USING NEURAL NETWORKS

P-647

PROJECT REPORT

Thesis submitted in partial fulfillment of the requirements for the

DEGREE OF

MASTER OF ENGINEERING IN

COMPUTER SCIENCE AND ENGINEERING

OF BHARATHIAR UNIVERSITY

BY

G.VIJAYA

Reg. No. 0037k0015

Guided By

Prof. Dr.S.Thangasamy Ph.D.,

Head of the Computer Science and Engineering Department

Department of Computer Science and Engineering

Kumaraguru College of Engineering

Coimbatore-641 006.

December- 2001

Department of Computer Science and Engineering

KUMARAGURU COLLEGE OF TECHNOLOGY

(Affiliated to the Bharathiar University)

Coimbatore – 641 006

CERTIFICATE

This is to certify that the project work entitled

TEXT FILE COMPRESSION BY USING NEURAL NETWORKS

Done by

G.VIJAYA
(Reg.No.0037K0015)

Submitted in partial fulfillment of the requirements for the award of the degree of

Master of Engineering In

Computer Science and Engineering

of Bharathiar University

[illegible] 20/12/01

Guide
Dr. S.THANGASAMY Ph.D.
Computer Science and Engineering Department
K.C.T., Coimbatore.

[illegible]

Head of the Department
Dr. S.THANGASAMY Ph.D.

Submitted for Viva – Voce examination held at

Kumaraguru College of technology on [illegible]

Internal examiner

External Examiner

DECLARATION

-, **G VIJAYA** hereby declare that this project work entitled "TEXT FILE COMPRESSION USING NEURAL NETWORKS" submitted to KUMARAGURU COLLEGE OF ENGINEERING , COIMBATORE(Affiliated to Bharathiar university) is a record of original work done by me under the supervision and guidance of Prof.Dr.S.Thangasamy Ph.D., Head,Department of computer science and Engineering.

Name of the candidate	Register Number	Signature of the candidate
G.VIJAYA	OO37K0015	[illegible] (G.VIJAYA)

Countersigned by : Staff in Charge [illegible]

Prof .Dr.S.Thangasamy Ph.D.,
Head of the Department
Department of Computer Science and Engineering
Kumaraguru College of Engineering
Coimbatore-641 006.

Place: Coimbatore

Date: [illegible]

ACKNOWLEDGEMENT

I express my deep gratitude to **Dr.Padmanaban Ph.D.,** Principal, Kumara Guru College of Engineering, Coimbatore for providing all facilities to carry out this project

With profound sense of gratitude and regards, I acknowledge with great pleasure the guidance and support extended by **Prof. Dr.S.Thangasamy Ph.D.,** Head of the department of Computer science and Engineering, Kumara guru College of Engineering, Coimbatore,for his valuable and continues guidance, suggestions, constructive criticisms and persistent encouragement in all phase of this project work. It had been a great pleasure to work under his guidance.

I express my deep and sincere thanks to all Computer science and Engineering staff for rendering timely help for successful completion of this project.

Last but least, I thank all my friends for their insistence co-operation and kind support to carry out this project successfully

SYNOPSIS

There are two main families in data compression techniques, loss less and lossy. Lossy compression allows a certain amount of loss of information in exchange for a greater compression ratio. These prove very good for compression of sound or image data where the loss of precision is not humanly noticeable. These algorithms also usually allow for adjustments for quality versus compression ratio.

Lossless compression on the other hand is focused on the exact recovery of the original data from the compressed data. These algorithms are used pretty much everywhere from backup software to network pack compression to military burst transmission.

Neural networks have the potential to extend data compression algorithms for text compression beyond the character level n-gram models now in use, but have usually been avoided because they are too slow to be practical. This method uses Limpel-Ziv compression algorithms (zip, gzip, compress) or Partial Prediction Method (PPM) or Burrows-Wheeler algorithms, currently the best known. The method which applies Limpel-Ziv algorithm gives a poor compression ratio but it is fast. The main reason for PPM methods not to achieve maximum compression is the prediction error in the modeling of probability distribution. Burrows-Wheeler algorithms produce superior compression but are significantly slower.

We introduce a model that produces better compression than popular methods which apply Limpel-Ziv compression algorithms and is competitive in time, space, PPM and Burrows-Wheeler algorithms, currently the best known. The compressor which uses a bit-level predictive arithmetic encoder using a 2 layer network, is fast (about 10^4 characters/second) because only 6 connections are simultaneously active and also it uses a variable learning rate optimized for one-pass training.

A system has been designed and implemented in c++ and tested.

CONTENTS

1. INTRODUCTION

There are two main families in data compression techniques, lossless and lossy. Lossy compression allows a certain amount of loss of information accuracy in exchange for a greater compression ratio. These prove very good for compression of sound or image data where the loss of precision is not humanly noticeable. These algorithms also usually allow for adjustments for quality versus compression ratio.

Lossless compression on the other hand is focused on the exact recovery of the original data from the compressed data. These algorithms are used pretty much everywhere from backup software to network pack compression to military burst transmission.

One of the motivations for using neural networks for data compression is , standard compression algorithms, such as Limpel-Ziv or Prediction by Partial Match (PPM) (Bell, Witten, and Cleary, 1989) or Burrows-Wheeler (Burrows and Wheeler, 1994) are based on simple n-gram models. These algorithms exploit the non-uniform distribution of text sequences found in most data. For example, the character trigram "*the*" is more common than "*qzv*" in English text, so the former would be assigned a shorter code. However, there are other types of learnable redundancies that cannot be modeled using n-gram frequencies. For example, Rosenfeld (1996) combined word trigrams with semantic associations, such as "*fire...heat*", where certain pairs of words are likely to occur near each other but the intervening text may vary, to achieve an unsurpassed word perplexity of 68, or about 1.23 bits per character (bpc), on the 38 million word Wall Street Journal corpus. Connectionist neural models (Feldman and Ballard, 1982) are well suited for modeling language constraints such as these, e.g. by using neurons to represent letters and words, and connections to model associations.

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

The rest of this thesis exposition is organized as follows: we begin by providing some minimal background information on text compression methods, revealing their strengths and weaknesses. Sections 1 describes the motivation behind this project, point out the goal that text compressors strive to accomplish, and interpret it as a pattern recognition problem. They provide the intuition of why and how we could get closer to our goal, and lead into some problems facing the existing methods. These problems are discussed in more detail in Section 2. Section 3 describes the proposed line of attak. Section 4 begins by combining the notion of re-representation with neural networks, and outlining a proposed avenue of getting text compression improvement. Section 5 discusses the test results. we finally conclude with a summary and suggestions for future work and further improvements in Section 6.

1.1 The Current Status Of the Problem Taken Up

The problem with the traditional statistical methods is that the minimum length of the pattern p is severely limited by implementation restrictions need to keep track of counts of all contexts, and so the probability estimate is not the most accurate one. The statistics may also be very dynamic and even adaptive methods may not be able to capture them efficiently. This suggests that alternative pattern recognition methods (such as neural networks) may be able to do a better job in modeling the probabilities by employing some form of "intelligent" learning. These alternative approaches have their own types of obstacles, however, which we need to overcome, in turn, in order to get any successful results. The main problem of interest that arises in present techniques, and is relevant to neural network applications is the problem of over-fitting, or using too much of freedom in the underlying model. The unnecessarily large model size eventually leads to memorizing all of the details in the sample data, while being unable to extract common patterns and generalize well outside of the sample set.

1.2 Relevance and Importance of the topic

Data compression is an important field of Computer Science mainly because of the reduced data communication and storage costs it achieves. Given the continued increase in the amount of data that needs to be transferred and/or archived now a days, the importance of data compression is unlikely to diminish in the foreseeable future. On the contrary, the great variety of data that allows for compression leads to the discovery of many new techniques specifically tailored to one type of data or another. The goal of this project is to concentrate on text compression, in particular, by pursuing an innovative and promising avenue for improving the compressibility of text. The main idea is that the standard ASCII representation is not necessarily the optimal way to order characters, and changing the representation of the alphabet may prove beneficial to various text processing tasks, including compression.

Current methods do not consider geometric information for prediction purposes. For instance, both letters *p* and *s* tend to predict the letter *h* (there are many words containing the sequences *ph* and *sh*). The letter *q*, however, tends to precede the letter *u* rather than *h*, and yet in the English alphabet (and in the ASCII code tables) *p* is closer to *q* than it is to *s*. Intuitively, any given character is endowed with the number of ``features" that affect what might be coming next. Examples of features include whether the character is alphabetic, small or capital, and whether it is a consonant. This leads us to build a (multidimensional) re-representation of the ASCII characters. One property that is intuitively desirable of such a re-representation is that characters that tend to precede the same characters are close under the new representation. That property would ensure that small changes in the contexts lead to small changes in the probability distributions, and we can restrict ourselves to considering only the class of such smooth transitions. Since neural networks are known to be good at learning and generalization of smooth data, they may be able to make predictions with higher confidence than the traditional methods. The algorithm that we propose, named **Text file compression**, is briefly summarized as follows: we first build the above-mentioned alphabet re-representation and use it in conjunction with a particular neural network model. At each step, the input context is re-represented and passed to the neural network which outputs the probability distribution of the next character given the particular context. The next character is then arithmetic-coded using the predicted probability distribution.

2.EXISTING MODELS

This section discusses the existing algorithms used for compressing text files and the advantages and disadvantages of each. It also discusses the requirements of a good compression algorithm.

There exist many data compression algorithms. Limpel-Ziv(LZ), PPM and a Block Sorted Lossless Algorithm.(M.Burrows and D.J.Wheeler)

2.1 Limpel Ziv

Limpel Ziv techniques exploit text redundancy caused by grammatical structure of English (or other language) text. Generally, they *pass/parse the input string into a sequence of sub strings, or dictionary words, and encode the text by replacing the passed token with its dictionary index or with a pointer to its last occurrence in the text. There are numerous variants that differ in the passing strategy, the token coding and the dictionary update heuristics (for adaptive additions and deletions). The main characteristics of all of them, and their biggest advantage is the fact that they are extremely fast and are therefore very suitable for practical purposes. Although they yield inferior over all compression compared to other algorithms. LZ-type algorithms are very popular because of their speed, and they are usually preferred choice for every day use.

2.2 Prediction By Partial Method

The PPM methods are the more successful methods in terms of compression efficiency. They generally gather (perhaps dynamically) some statistical information about the probability distribution source (the file to be executed). This information is then used to estimate the probability distribution of the next character given what the previous n characters were, and finally the probability estimate is used in conjunction with an entropy coder to encode the next character. The probability of each character is approximated by the fraction of times this character occurred after the particular context of length n. The PPM methods are also called context methods and character prediction method because; they basically predict the probability distribution of the next character in the given context.

The main reason for PPM methods not to achieve maximum compression is the prediction error in the modeling of the probability distribution. PPM methods are much slower than LZ algorithms.

2.3 A Block-Sorting Lossesless Data Compression Algorithm

(M.Burrows and D.J.Wheeler)

The most widely used data compression algorithms are based on sequential data compressors of the Limpel-Ziv. Statistical modeling techniques may produce superior compression, but are significantly slower.

This algorithm does not process its input sequentially, but instead processes a block of text as a single unit. The idea is to apply a reversible transformation to a block of text a single unit to form a new block that contain the same characters, but easier to compress by simple compression algorithms. The transformation tends to group characters together. The probability of finding a character close to another instance of the same character is increased substantially.

This algorithm transforms a string 's' of N characters by forming the rotation (cyclic shifts) of 's', sorting them lexicographically, and extracting the last character of each of rotations. A string 'l' is formed from these characters, where the 'i'th character of 'l' is the last character of 'i'th sorted rotation. In addition to 'l', the algorithm computes the index of the original string 's' in the sorted list of rotations. This is an efficient algorithm to compute the original string 's', given only 'l' and 'i'.

The sorting operation brings together rotations with the same initial character. Since the initial character of rotations are adjacent to the final characters, consecutive characters in ‘l’ are adjacent to similar strings in ’s’. If the context of a character is a good predictor for the character ’s’ will be easy to compress with a simple locally adaptive compression algorithm.

2.4 Conditions For Data Compression

When transferring information, the choice of the data representation determines how fast the transfer is preformed. A judicious choice can improve the throughput of a transmission channel without changing the channel itself.

Various data compression techniques attempt to minimize the average codeword length by devising an optimal code that depends on the probability P with which a symbol is being used. If a symbol is issued infrequently it can be assigned a long codeword. For frequently issued symbols, very short encoding are more useful in compression.

Restriction imposed on the prospective codes.

1. Each code word corresponds to exactly one symbol;
2. Decoding should not require any look ahead. After reading each symbol it should be possible to determine whether the end of a string encoding a symbol of the original message has been reached. A code matching this requirement is called a code with the prefix property, and it means that no codeword is a prefix of another codeword therefore; no special punctuation is required to separate two code words in a coded message.
3. The length of the codeword for a given symbol mj should not exceed the length of the codeword of a less probable symbol; that is if $p(mj) \leq p(mj)$,

   then $l(mi) >= l(mj)$ for $1 \leq i, j \leq n$..

4. In an optimal encoding system, there should not be any unused short codeword either as stand-alone encoding or as prefixes for longer codeword, since this would mean that longer code words were created unnecessarily
5. Compressions ratio is defined as the ratio of the size or rate of the original data to the size or rate of the compressed data.

PROPOSED LINE OF ATTACK

3.PROPOSED LINE OF ATTACK

The objective of the project is to improve the compression ratio by using neural network along with arithmetic coding. Text file compression has two parts first one is the predictor and the second one is the arithmetic encoder. Arithmetic encoder peeks at the next character and assigns a code based on the probability assigned to it by the predictor. The idea is to assign the shorter codes for the most likely characters, so that the output will be shorter on average

```
Compression                              P(a) = .04
                    +-----------+        P(b) = .003+---------+
 the cat in th_  -->| Predictor |-->...         -->| Encoder |--> X
                    +-----------+        P(e) = .3  +---------+    |
                                           ...           ^         |
                                                         e --+     |
                                                                   |
                                                         +---------+
Decompression                            P(a) = .04      v
                   +-----------+         P(b) = .003  +---------+
the cat in th_  -->| Predictor | -->  ...        --> | Decoder | --> e
                ^  +-----------+         P(e) = .3    +---------+    |
                |                          ...
                +----------------------------------------------------+
```

During decompression, the predictor makes an identical series of probability estimates for each character. The decoder does the opposite of the encoder. It matches the code to what the encoder would have assigned to each character in order to recover the original character.

DETAILS OF THE PROPOSED METHODLOGY

4. DETAILS OF THE PROPOSED METHODOLOGY

This section discusses in detail the concepts of neural network, Arithmetic coding ,Hash function and Bit operators

4.1 Concepts Of Neural Networks

4.1.1 Real Neurons

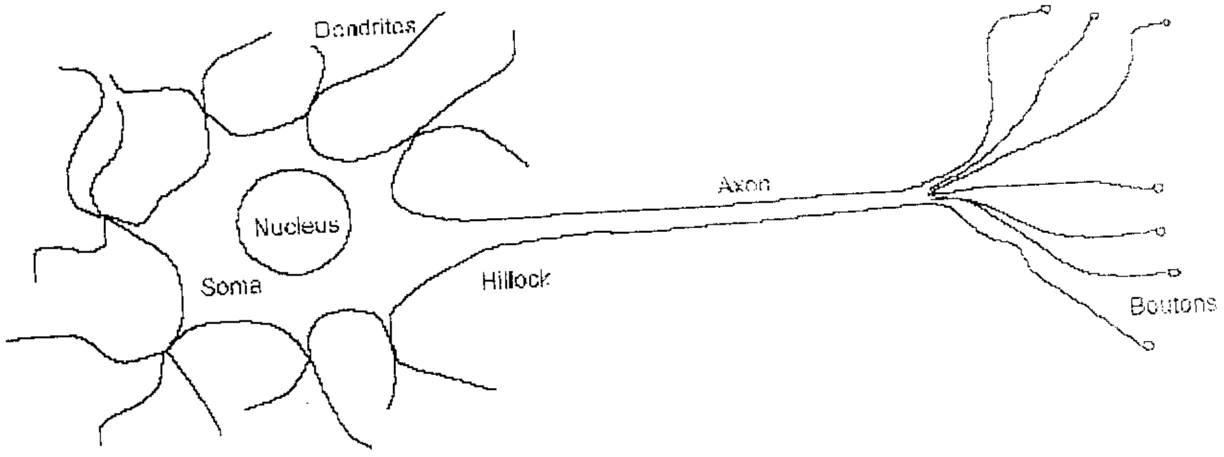


Figure 1. A Biological Neuron

A neuron operates by receiving signals from other neurons through connections, called **synapses**. The combination of these signals, in excess of a certain **threshold or activation** level, will result in the neuron **firing**, that is sending a signal on to other neurons connected to it. Some signals act as **excitations** and others as **inhibitions** to a neuron firing. **What we call thinking is believed to be the collective effect of the presence or absence of firings in the pattern of synaptic connections between neurons.**

This sounds very simplistic until we recognize that there are approximately one hundred billion (100,000,000,000) neurons each connected to as many as one thousand (1,000) others in the human brain. The massive number of neurons and the complexity of their interconnections result in a "thinking machine", your brain.

Each neuron has a body, called the **soma**. The soma is much like the body of any other cell. It contains the cell nucleus, various bio-chemical factories and other components that support ongoing activity.

Surrounding the soma are **dendrites**. The dendrites are receptors for signals generated by other neurons. These signals may be excitatory or inhibitory. All signals present at the dendrites of a neuron are combined and the result will determine whether or not that neuron will fire. If a neuron fires, an electrical impulse is generated. This impulse starts at the base, called the **hillock**, of a long cellular extension, called the **axon**, and proceeds down the axon to its ends.

The end of the axon is actually split into multiple ends, called the **boutons**. The boutons are connected to the dendrites of other neurons and the resulting interconnections are the previously discussed synapses. (Actually, the boutons do not touch the dendrites; there is a small gap between them.) If a neuron has fired, the electrical impulse that has been generated stimulates the boutons and results in electrochemical activity, which transmits the signal across the synapses to the receiving dendrites.

At rest, the neuron maintains an electrical potential of about 40-60 millivolts. When a neuron fires, an electrical impulse is created which is the result of a change in potential to about 90-100 millivolts. This impulse travels between 0.5 to 100 meters per second and lasts for about 1 millisecond. Once a neuron fires, it must rest for several milliseconds before it can fire again. In some circumstances, the repetition rate may be as fast as 100 times per second, equivalent to 10 milliseconds per firing.

Compare this to a very fast electronic computer whose signals travel at about 200,000,000 meters per second (speed of light in a wire is 2/3 of that in free air), whose impulses last for 10 nanoseconds and may repeat such an impulse immediately in each succeeding 10 nanoseconds continuously. Electronic computers have at least a 2,000,000 times advantage in signal transmission speed and 1,000,000 times advantage in signal repetition rate.

It is clear that if signal speed or rate were the sole criteria for processing performance, electronic computers would win hands down. What the human brain lacks in these, it makes up in numbers of elements and interconnection complexity between those elements. This difference in structure manifests itself in at least one important way; the human brain is not as quick as an electronic computer at arithmetic, but it is many times faster and hugely more capable at recognition of patterns and perception of relationships.

The human brain differs in another, extremely important, respect beyond speed: it is capable of "self-programming" or adaptation in response to changing external stimuli. In other words, it can learn. The brain has developed ways for neurons to change their response to new stimulus patterns so that similar events may affect future responses. In particular, the sensitivity to new patterns seems more extensive in proportion to their importance to survival or if they are reinforced by repetition.

4.1.2 Neural Network Structure

Neural networks are models of biological neural structures. The starting point for most neural networks is a model neuron, as in Figure 2. This neuron consists of multiple inputs and a single output. Each input is modified by a **weight**, which multiplies with the input value. The neuron will combine these weighted inputs and, with reference to a threshold value and activation function, use these to determine its output. This behavior follows closely our understanding of how real neurons work.

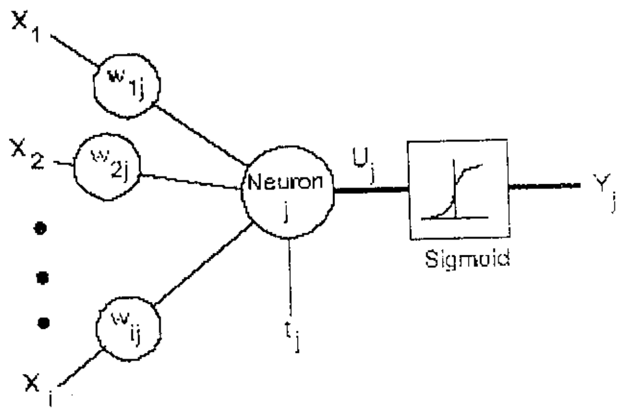


Figure 2. A Model Neuron

While there is a fair understanding of how an individual neuron works, there is still a great deal of research and mostly conjecture regarding the way neurons organize themselves and the mechanisms used by arrays of neurons to adapt their behavior to external stimuli. There are a large number of experimental neural network structures currently in use reflecting this state of continuing research.

In our case, we will only describe the structure, mathematics and behavior of that structure known as the **back propagation network**. This is the most prevalent and generalized neural network currently in use. To build a back propagation network, proceed in the following fashion. First, take a number of neurons and array them to form a **layer**. A layer has all its inputs connected to either a preceding layer or the inputs from the external world, but not both within the same layer. A layer has all its outputs connected to either a succeeding layer or the outputs to the external world, but not both within the same layer.

Next, multiple layers are then arrayed one succeeding the other so that there is an input layer, multiple intermediate layers and finally an output layer, as in Figure 3. Intermediate layers, that are those that have no inputs or outputs to the external world, are called **hidden layers**. Back propagation neural networks are usually **fully connected**. This means that each neuron is connected to every output from the preceding layer or one input from the external world if the neuron is in the first layer and, correspondingly, each neuron has its output connected to every neuron in the succeeding layer.

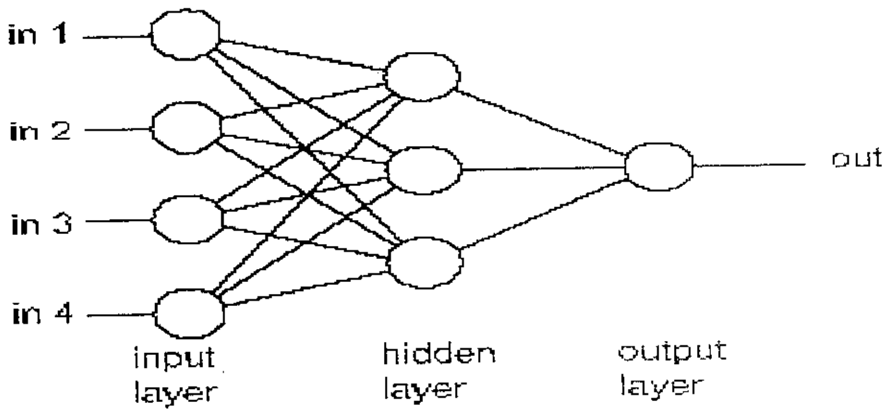


Figure 3. Back propagation Network

The input layer is considered a distributor of the signals from the external world. Hidden layers are considered to be categorizers or feature detectors of such signals. The output layer is considered a collector of the features detected and producer of the response. While this view of the neural network may be helpful in conceptualizing the functions of the layers, you should not take this model too literally as the functions described may not be so specific or localized.

With this picture of how a neural network is constructed, we can now proceed to describe the operation of the network in a meaningful fashion.

4.1.3 Neural Network Operation

The output of each neuron is a function of its inputs. In particular, the output of the *j*th neuron in any layer is described by two sets of equations:

$$[U_j = \Sigma (X_j . W_{ij})]$$

and

$$[Y_j = F_{th} (Uj + tj)]$$

For every neuron, *j*, in a layer, each of the ***i*** inputs, X_i, to that layer is multiplied by a previously established weight, w_{ij}. These are all summed together, resulting in the internal value of this operation, U_j. This value is then biased by a previously established threshold value, t_j, and sent through an activation function, F_{th}. This activation function is usually the sigmoid function, which has an input to output mapping as shown in Figure below. The resulting output, Y_j, is an input to the next layer or it is a response of the neural network if it is the last layer.

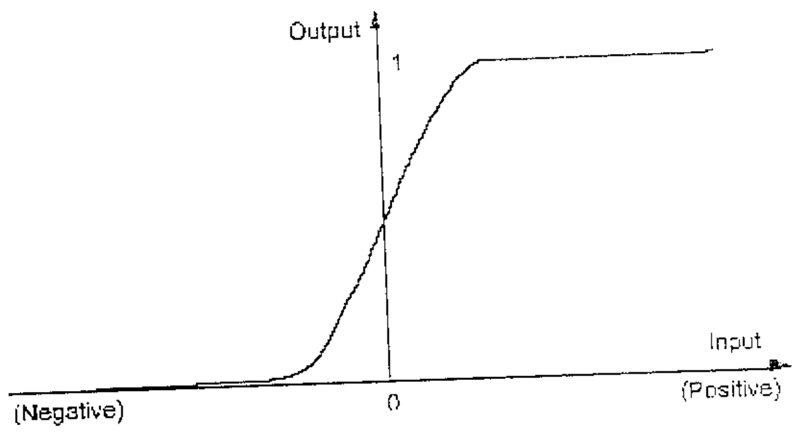


Figure 4. Sigmoid Function

In essence, Equation 1 implements the combination operation of the neuron and Equation 2 implements the firing of the neuron.

From these equations, a predetermined set of weights, a predetermined set of threshold values and a description of the network structure (that is the number of layers and the number of neurons in each layer), it is possible to compute the response of the neural network to any set of inputs.

4.1.4 Neural Network Learning

Learning in a neural network is called **training**. Like training in athletics, training in a neural network requires a coach, someone that describes to the neural network what it should have produced as a response. From the difference between the desired response and the actual response, the **error** is determined and a portion of it is propagated backward through the network. At each neuron in the network the error is used to adjust the weights and threshold values of the neuron, so that the next time, the error in the network response will be less for the same inputs.

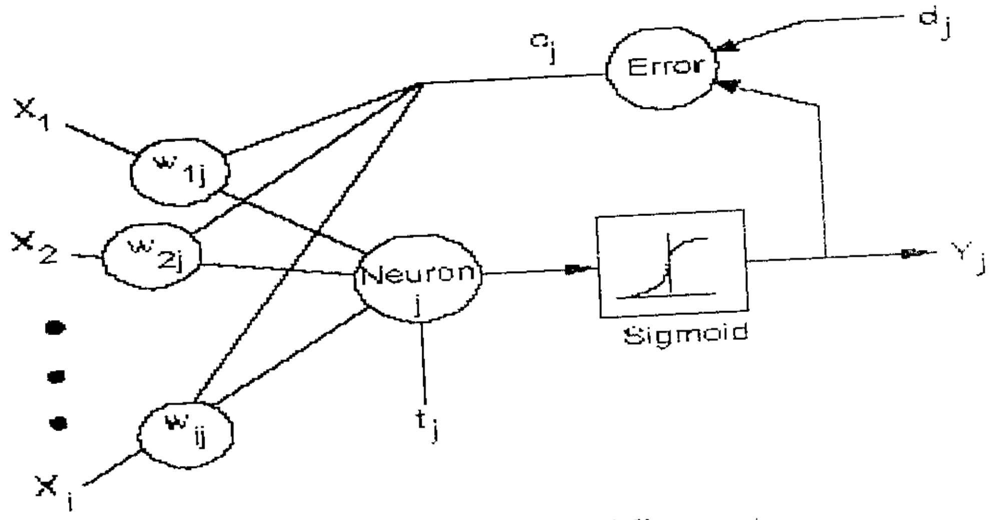


Figure 5. Neuron Weight Adjustment

This corrective procedure is called **back propagation** (hence the name of the neural network) and it is applied continuously and repetitively for each set of inputs and corresponding set of outputs produced in response to the inputs. This procedure continues so long as the individual or total errors in the responses exceed a specified level or until there are no measurable errors. At this point, the neural network has learned the training material and you can stop the training process and use the neural network to produce responses to new input data.

Back propagation starts at the output layer with the following equations:

$$W_{ij}=W'_{ij}+LR\,.\,e_j\,.X_i$$

and

$$e_j = Y_j.\,(1 - Y_j)\,.\,(d_j - Y_j)$$

For the *i*th input of the *j*th neuron in the output layer, the weight w_{ij} is adjusted by adding to the previous weight value, w'_{ij}, a term determined by the product of a ***learning*** *rate*, ***LR***, an error term, e_j, and the value of the *i*th input, X_i. The error term, e_j, for the jth the product of the actual output, Yj, its complement, 1 - Yj, and the difference between the desired output, dj, and the actual output determines neuron.

Once the error terms are computed and weights are adjusted for the output layer, the values are recorded and the next layer back is adjusted. The same weight adjustment process, determined by Equation 3, is followed, but the error term is generated by a slightly modified version of Equation 4. This modification is:

$$e_j = Y_j \,.\, (1 - Y_j) \,.\, \Sigma\, (e_k - W'_{jk})$$

In this version, the difference between the desired output and the actual output is replaced by the sum of the error terms for each neuron, *k*, in the layer immediately succeeding the layer being processed (remember, we are going backwards through the layers so these terms have already been computed) times the respective pre-adjustment weights.

The learning rate, ***LR***, applies a greater or lesser portion of the respective adjustment to the old weight. If the factor is set to a large value, then the neural network may learn more quickly, but if there is a large variability in the input set then the network may not learn very well or at all. In real terms, setting the learning rate to a large value is analogous to giving a child a spanking, but that is inappropriate and counter-productive to learning if the offense is so simple as forgetting to tie their shoelaces. Usually, it is better to set the factor to a small value and edge it upward if the learning rate seems slow.

In many cases, it is useful to use a revised weight adjustment process. This is described by the equation:

$$W_{ij} = W'_{ij} + (1 - M) \,.\, LR.e_j \,.\, X_i + M \,.\, (W'_{ij} - W''_{ij})$$

This is similar to equation 3, with a **momentum** factor, ***M***, the previous weight, w'_{ij}, and the next to previous weight, w''_{ij}, included in the last term. This extra term allows for momentum in weight adjustment. Momentum basically allows a change to the weights to persist for a number of adjustment cycles. The magnitude of the persistence is controlled by the momentum factor. If the momentum factor is set to 0, then the equation reduces to that of Equation 3. If the momentum factor is increased from 0, then increasingly greater persistence of previous adjustments is allowed in modifying the current adjustment. This can improve the learning rate in some situations, by helping to smooth out unusual conditions in the training set.

As you train the network, the total error, that is the sum of the errors over all the training sets, will become smaller and smaller. Once the network reduces the total error to the limit set, training may stop. You may then apply the network, using the weights and thresholds as trained.

It is a good idea to set aside some subset of all the inputs available and reserve them for **testing** the trained network. By comparing the output of a trained network on these test sets to the outputs you know to be correct, you can gain greater confidence in the validity of the training. If you are satisfied at this point, then the neural network is ready for **running**.

Usually, no back propagation takes place in this running mode as was done in the training mode. This is because there is often no way to be immediately certain of the desired response. If there were, there would be no need for the processing capabilities of the neural network! Instead, as the validity of the neural network outputs or predictions are verified or contradicted over time, you will either be satisfied with the existing performance or determine a need for new training. In this case, the additional input sets collected since the last training session may be used to extend and improve the training data.

4.1.5 Limitations Of Back Propagation

The most serious drawback of using back propagation is that it is inefficient. This is more commonly know as the scaling problem where as the size of the network increases, the network becomes more computationally intensive, and the time require to train the network grows exponentially. For this reason we are using only two layer network .It learns during one pass itself.

Many say that the speed of the back propagation routine is a major drawback, but I consider this to be a minor drawback only since the speed of new computers doubles every third year and speed becomes a less important issue than it is today. We can make it fast by selecting the parameters and by using hash functions

A careful selection of step size is often necessary to ensure smooth convergence. Large step size may cause the network to become paralyzed. When network paralysis occurs, further training does little for convergence. On the other hand, if the size is too small, convergence can be very slow.

Back propagation searches on the error surface along the gradient in order to minimize the error criterion. It is likely to get stuck in a local minimum.

The process of weights based on the gradients is repeated until a minimum is reached. There are several stopping criteria that can be considered. Based in the error to be minimized and based on the gradient. If these two criteria are sensitive to the choice of parameters and may lead to poor results if the parameters are improperly chosen.

In order to recognize new patterns, the network needs to be trained with these patterns along with previously known patterns .If only new patterns are provided for retraining, and then old patterns may be forgotten. To over come this drawback we use two parameters called short-term learning and long-term learning. so that it can recognize both the old and new patterns.

Another limitation is that back propagation network is prone to local minima, just like any other gradient descent algorithm

4.2 Concepts Of Arithmetic Coding

4.2.1 Arithmetic coding

Arithmetic coding is entropy coder widely used, the only problem is it's speed, but compression tends to be better than Huffman can achieve. The idea behind arithmetic coding is to have a probability line, 0-1, and assign to every symbol a range in this line based on its probability, the higher the probability, the higher range that assigns to it. Once we have defined the ranges and the probability line, start to encode symbols; every symbol defines where the output floating point number lands.

For example

Symbol	Probability	Range
A	2	[0.0, 0.5)
B	1	[0.5, 0.75)
C	1	[0.7.5, 1.0)

Note that the "[" means that the number is also included, so all the numbers from 0 to 5 belong to "a" but 5. And then we start to code the symbols and compute our output number. The algorithm to compute the output number is:

- Low = 0
- High = 1
- Loop. For all the symbols
    - Range = high - low
    - High = low + range * high range of the symbol being coded
    - Low = low + range * low range of the symbol being coded

Where

Range, keeps track of where the next range should be and

High and low, specify the output number.

And now let's see an example

Symbol	Range	Low value	High value
		0	1
B	1	0.5	0.75
A	0.25	0.5	0.625
C	0.125	0.59375	0.625
A	0.03125	0.59375	0.609375

The output number will be 0.59375. The way of decoding is first to see where the number lands, output the corresponding symbol, and then extract the range of this symbol from the floating-point number. The following is the algorithm for extracting the ranges

- Loop. For all the symbols.
    - Range = high range of the symbol - low range of the symbol
    - Number = number - low range of the symbol
    - Number = number / range

And this is how decoding is performed

Symbol	Range	Number
B	0.25	0.59375
A	0.5	0.375
C	0.25	0.75
A	0.5	0

Symbol	Probability	Range
A	2	[0.0, 0.5)
B	1	[0.5, 0.75)
C	1	[0.75, 1.0)

You may reserve a little range for an Elf symbol, but in the case of an entropy coder you'll not need it (the main compressor will know when to stop), with and stand-alone code you can pass to the decompressor the length of the file, so it knows when to stop.

4.2.2 Implementation

As you can see from the example it is a must that the whole floating point number is passed to the decompressor, no rounding can be performed, but with the today's FPU (Floating Point Unit) the higher precision which it can offer is 80 bits, so we can't work with the whole number. So instead we'll need to redefine our range, instead of 0-1 it will be 0000h to FFFFh , which in fact is the same. And we'll also reduce the probabilities so we don't need the whole part, only 16 bits. It is the same.

For example the following table gives the hexadecimal values for decimal values.

0.000	0.250	0.500	0,750	1.000
0000h	4000h	8000h	C000h	FFFFh

If we take a number and divide it by the maximum (FFFFh) will clearly see it

- 0000h: 0/65535 = 0,0
- 4000h: 16384/65535 = 0,25
- 8000h: 32768/65535 = 0,5
- C000h: 49152/65535 = 0,75
- FFFFh: 65535/65535 = 1,0

Adjust the probabilities so the bits needed for operating with the number aren't above 16 its. And now, once defined a new interval, and are sure that work with only 16 bits, start to do it. They way we deal with the infinite number is to have only loaded the 16 first bits, and when needed shift more on to it
1100 0110 0001 000 0011 0100 ...
As new bits are needed they'll be shifted. The algorithm of arithmetic coding makes that if ever the MSB of both high and low match are equal, then they'll never change, this is how we can output the higher bits of the output infinite number, and continue working with just 16 bits. However this is not always the case.

4.2.3 Underflow

Underflow occurs when both high and low get close to a number but theirs MSB don't match. High = 0,300001 and Low = 0,29997 if we ever have such numbers, and the continue getting closer and closer we'll not be able to output the MSB, and then in a few iterations our 16 bits will not be enough, what we have to do in this situation is to shift the second digit (in our implementation the second bit) and when finally both MSB are equal also output the digits discarded.

4.2.4 Gathering the Probabilities

In this example we'll use a simple statically order-0 model. You have an array initialized to 0, and you count there the occurrences of every byte. And then once we have them we have to adjust them, so they don't make us need more than 16 bits in the calculations, if we want to accomplish that, the total of our probabilities should be below 16,384. (2^14). To scale them we divide all the probabilities by a factor till all of them fit in 8 bits, however there's an easier (and faster) way of doing so, you get the maximum probability, divide it by 256, this is the factor that you'll use to scale the probabilities. Also when dividing, if the result ever is 0 or below put it to one, so the symbol has a range. The next scaling deals with the maximum of 2^14, add to a value (initialized to 0) all the probabilities, and then check if it's above 2^14, it is then divide them by a factor, (2 or 4) and the following assumptions will be true:

- All the probabilities are inside the range of 0-255. This helps saving the header with the probabilities.
- The addition of all of them doesn't get above 2^14, and thus we'll need only 16 bits for the computations.

4.2.5 Saving The Probabilities

Our probabilities are one byte long, so we can save the whole array, as a maximum it can be 256 bytes, and it's only written once, so it will not hurt compression a lot. If you expect some symbols to not appear you could   code it. If you expect some probabilities to have lower values than others, you can use a flag to say how many bits the next probability uses, and then code one with 4 or 8 bits, anyway you should tune the parameters.

Assign Ranges

For every symbol we have to define its high value and low value, they define the range, doing this is rather simple, we use its probability

Symbol	Probability	Low	High	0-1 (x/4)
A	2	0	2	[0.0, 0.5)
B	1	2	3	[0.5, 0.75)
C	1	3	4	[0.75, 1)

What we'll use is high and low, and when computing the number we'll perform the division, to make it fit between 0 and 1. Anyway if you have a look at high and low you'll notice that the low value of the current symbol is equal to the high value of the last symbol, we can use it to use half the memory, we only have to care about setting –1 symbol with a high value of 0.

Symbol	Probability	High
-1	0	0
A	2	2
B	1	3
C	1	4

And thus when reading the high value of a symbol we read it in its position, and for the low value we read the entry "position 1".

Assign to the high value the current probability of the symbol + the last high value, and set it up with the symbol "-1" with a high probability of 0. I.e.: When reading the range of the symbol "b" we read its high value at the current position (of the symbol in the table) "3" and for the low value, the previous "2". And because our probabilities take one byte, the whole table will only take 256 bytes.

4.2.6 Pseudocode

The following is the pseudo code for initialization:

- Get probabilities and scale them

- Save probabilities in the output file
- High = FFFFh (16 bits)
- Low = 0000h (16 bits)
- Underflow bits = 0 (16 bits should be enough)

Where:

- High and low, they define where the output number falls.
- Underflow bits, the bits which could have produced underflow and thus they were shifted.

And the routine to encode a symbol:

- Range = (high - low) + 1
- High = low + ((range * high values [symbol]) / scale) - 1
- Low = low + (range * high values [symbol - 1]) / scale
- Loop. (Will exit when no more bits can be outputted or shifted)
- MSB of high = MSB of low?
- Yes
  - Output MSB of low
  - Loop. While underflow bits > 0 Let's output underflow bits pending for output
    - Output Not (MSB of low)
  - Go to shift
- No
  - Second MSB of low = 1 and Second MSB of high = 0? Check for underflow
  - Yes
    - Underflow bits += 1 Here we shift to avoid underflow
    - Low = low & 3FFFh
    - High = high | 4000h

    - Go to shift
  - No
    - The routine for encoding a symbol ends here.

Shift:

Shift low to the left one time. Put in low and high new bits

- Shift high to the left one time, and or the lbs. with the value 1
- Repeat to the first loop.

Explanations:

- Note that the formulae before the loop should be done with 32 bit precision. (Word, long)
- MSB of high means the following, with a 16 bits number like that abbb bbbb bbbb bbbb , a is the MSB bit of it.
- Not (MSB of low) is "Bit wise complement operator" in C used in the following way: ~low in asm is just "not ax". First you perform not on low and then you output its MSB bit.
- "&" Means "bit wise and".
- "|" Means "bit wise inclusive or". (Or)
- Range must be 32 bits long, because the formula needs this precision.
- Scale is the addition of all the probabilities.

Once we have encoded all the symbols we have to flush the encode (output the last bits) output the second MSB of low and also underflow_bits+1 in the way we outputted underflow bits. Because our maximum number of bits is 16 we also have to output 16 bits all of them 0) so the decoder will get enough bytes.

Decoding

The first thing to do when decoding is read the probabilities, because the encode did the scaling we just have to read them and to do the ranges.

The process will be the following:

See in what symbol our number falls, extract the code of this symbol from the code. Before starting we have to init "code" this value will hold the bits from the input, init it to the first 16 bits in the input. And this is how it's done.

- Range = (high - low) + 1 See where the number lands
- Temp = ((code - low) + 1) * scale) - 1) / range)
- See what symbols corresponds to temp.
- Range = (high - low) + 1 Extract the symbol code
- High = low + ((range * high values [symbol]) / scale) - 1
- Low = low + (range * high values [symbol - 1]) / scale Note that those formulae are the same that the encoder uses
- Loop.
- MSB of high = MSB of low?
- Yes
    - Go to shift
- No
    - Second MSB of low = 1 and Second MSB of high = 0?
    - Yes
        - Code = code ^ 4000h
        - Low = low & 3FFFh
        - High = high | 4000h
        - Go to shift
    - No
        - The routine for decoding a symbol ends here.

Shift:

· Shift low to the left one time.

· Shift high to the left one time, and or the lbs. with the value 1

TEXT FILE COMPRESSION

· Shift code to the left one time, and or it the next bit in the input

· Repeat to the first loop.

When searching for the current number (temp) in the table we use a for loop, which based in the fact that the probabilities are sorted from low to high, have to do one comparison in the current symbol, until it's in the range of the number.

4.3 Text File Compression

4.3.1 Role Of Neural Network

Neural network does the prediction. Predictor predicts the next bit , given the bits so far. Input to the neural network is the bits.

4.3.1.1 A Maximum Entropy Neural Network

In a predictive encoder, interested in predicting the next input symbol y, given a feature vector **x** = $x_1, x_2,\ldots, x_M$, where each x_i is a 1 if a particular context is present in the input history, and 0 otherwise.
In this implementation, it predicts one bit at a time, so y is either 0 or 1. Estimate P(y = 1|x_i) » N(1)/N, for each context x_i, by counting the number of times, N(1), that y = 1 occurs in the N occurrences of that context. In an n-gram character model (n about 4 to 6), there would be one active input (x_i = 1) for each of the n contexts.

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

1. $P(\mathbf{x}, y) = 1/Z \; P_i \; a_i^{f_i(x,y)}$

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

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

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

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

3. $P(y|\mathbf{x}) = g(S_i w_i x_i)$, where
4. $g(x) = 1/(1 + e^{-x})$

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

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

5. $w_i = w_i + Dw_i$
6. $Dw_i = hx_iE$

Where h is the learning rate, usually set *ad hoc* to around 0.1 to 0.5. This is fine for batch mode training, but for on-line compression, where each training sample is presented .

4.3.1.2Maximum Entropy ON-Line Learning

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

7. $p \,^\circ\, P(y|x_i = 1) = g(w_i)$
8. $w_i = g^{-1}(p) = \ln p/(1 - p) \gg \ln N(1)/N(0)$

Where N(1) and N(0) are the number of times y has been observed to be a 1 or 0 respectively in context x_i.

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

9. $Dw_i = \ln (N(1)+1)/N(0) - \ln N(1)/N(0)$ » $1/N(1)$

and if y = 0,

10. $Dw_i = \ln N(1)/(N(0)+1) - \ln N(1)/N(0)$ » $-1/N(0)$

Find h such that the learning equation $Dw_i = hx_iE$ (which converges whether or not the inputs are independent) is optimal for independent inputs. For the case $x_i = 1$,

11. $Dw_i = Dw_iE/E = hE$
12. $h = Dw_i/E$

If y = 1, then

13. $E = 1 - p = 1 - N(1)/N = N(0)/N$
14. $h = Dw_i/E = 1/EN(1) = N/N(0)N(1) = 1/s^2$

And if y = 0, then

15. $E = -p = -N(1)/N$
16. $h = Dw_i/E = -1/EN(0) = N/N(0)N(1) = 1/s^2$

Where ,

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

If the inputs are correlated, then using the output error to adjust the weights still allows the network to converge, but not at the optimal rate. In the extreme case, if there are m perfectly correlated copies of x_i, and then the total effect will be equal to a single input with weight mw_i. The optimal learning rate will be $h = 1/ms^2$ in this case. Since the correlation cannot be determined locally, we introduce a parameter h_L, called the long term learning rate, where 1/m £ h_L £ 1. The weight update equation then becomes

17. $Dw_i = h_L E/s^2$

In addition, we have assumed that the data is stationary, that p = P(y|x_i) does not change over time. If the probability were fixed, It is clear that the last few bits of input history is a better predictor of the next bit than simply weighting all of the bits equally, as h_L does. The solution is to impose a lower bound, h_S, on the learning rate, a parameter that we call the ***short term*** learning rate. This has the effect of weighting the last $1/h_S$ bits more heavily.

18. $Dw_i = (h_S + h_L/s^2)E$

Finally, recall that s^2 = N(0)N(1)/N = Np(1 - p), which means that the learning rate is undefined when either count is 0, or equivalently, when p = N(1)/N is 0 or 1. A common solution is to add a small offset, d, to each of the counts N(0) and N(1). The value of d is d = 0.5.

Thus in (18),

$s^2 = (N+2d) / (N(0)+d)(N(1)+d)$

h should choose more carefully.

4.3.1.3 Bit Operations

Bit operations can be used for masking or eliminating selected bits from an operand. AND, EXCULSIVE OR, INCLUSIVE OR, LEFT SHIFT, RIGHT SHIFT and COMPLEMENT are the operators used for handling bits.

4.3.1.4 Hash Function By Division Method

The simplest of all the methods of hashing an integer *x* is to divide *x* by *M* and then to use the remainder modulo *M*. This is called the ***division method of hashing***. In this case, the hash function is

$$h(x)=|x| \bmod M.$$

Generally, this approach is quite good for just about any value of *M*. However, in certain situations some extra care is needed in the selection of a suitable value for *M*. For example, it is often convenient to make *M* an even number. But this means that *h*(*x*) is even if *x* is even; and *h*(*x*) is odd if *x* is odd. If all possible keys are equi probable, then this is not a problem. However if even keys are more likely than odd keys, the function

$$h(x)=x \bmod M$$

will not spread the hashed values of those keys evenly.

Similarly, it is often tempting to let *M* be a power of two. For example, for some integer *k*>1. In this case, the hash function h(x)=x mod 2^k simply extracts the bottom *k* bits of the binary representation of *x*. While this hash function is quite easy to compute, it is not a desirable function because it does not depend on all the bits in the binary representation of *x*.

For these reasons *M* is often chosen to be a prime number. For example, suppose there is a bias in the way the keys are created that makes it more likely for a key to be a multiple of some small constant, say two or three. Then making *M* a prime increases the likelihood that those keys are spread out evenly. Also, if *M* is a prime number, the division of *x* by that prime number depends on all the bits of *x*, not just the bottom *k* bits, for some small constant *k*.

The division method is extremely simple to implement. An advantage of the division method is that M need not be a compile-time constant--its value can be determined at run time. In any event, the running time of this implementation is clearly a constant.

A potential disadvantage of the division method is due to the property A potential disadvantage of the division method is due to the property that consecutive keys map to consecutive hash values:

h(i)=i

h(i+1) = i+1 (mod M)

h(i+2) = i+2 (mod M)

While this ensures that consecutive keys do not collide, it does mean that consecutive array locations will be occupied.
This project uses the hash function, is a prime number which is below 2^22-2^16.Hash function is used to map the huge set of context into a smaller set of inputs. The input to the hash function is the context, treated as a number, and the output is the corresponding unit.

4.3.1.5 Process Design

Compression

The file listed are compressed and stored in the archive, which is created. The archive must not already exist. File names may specify a path, which is stored. If there are no file names on the command line, then file compression prompts for them, reading until the first blank line or end of the file.

Decompression

No file names are specified. The archive must exist. If a path is stored, the file is extracted to the appropriate directory, which must exist. File compression does not create directories. IF the file to be extracted already exist, it is not replaced: rather it is compared with the archive file , and the offset of the first difference is reported. It is not possible to add, remove, or update files in an existing archive.
The archive file names are stored in a readable header as below:
File compression \ r \ n

Size name \ r \ n

Size name \ r \ n

\ 032 \ f \0

Where “size” is the original number of bytes, as a 9 digit decimal number, right justified With leading spaces,” name” is the file name, \ r is the carriage return, \n is the line feed, \f is form feed, \ 032 is the DOS end of file character, \ 0 is a NULL.

The header is followed by the compressed data in binary format. The input files are concatenated and treated as a single data stream.

Data is compressed using a bit stream predictive arithmetic encoder with a neural network predictor. The predictor estimates the probabilities P (0) + P (1) = 1 for each bit, given the previous bits. The arithmetic encoder begins with a range [0,1] , and divides the range into two sub ranges for each input bit, with sizes proportional to P(0) and P(1).

The output is the shortest number that can be expressed as a base 256 fraction (MSB first) within the resulting range.

The decompressor makes the same series of subdivisions using predictions P(0) and P(1), given the data decompressed so far. It examines the compressed data to select 0 or 1 according to which range it is in, and outputs that bit.

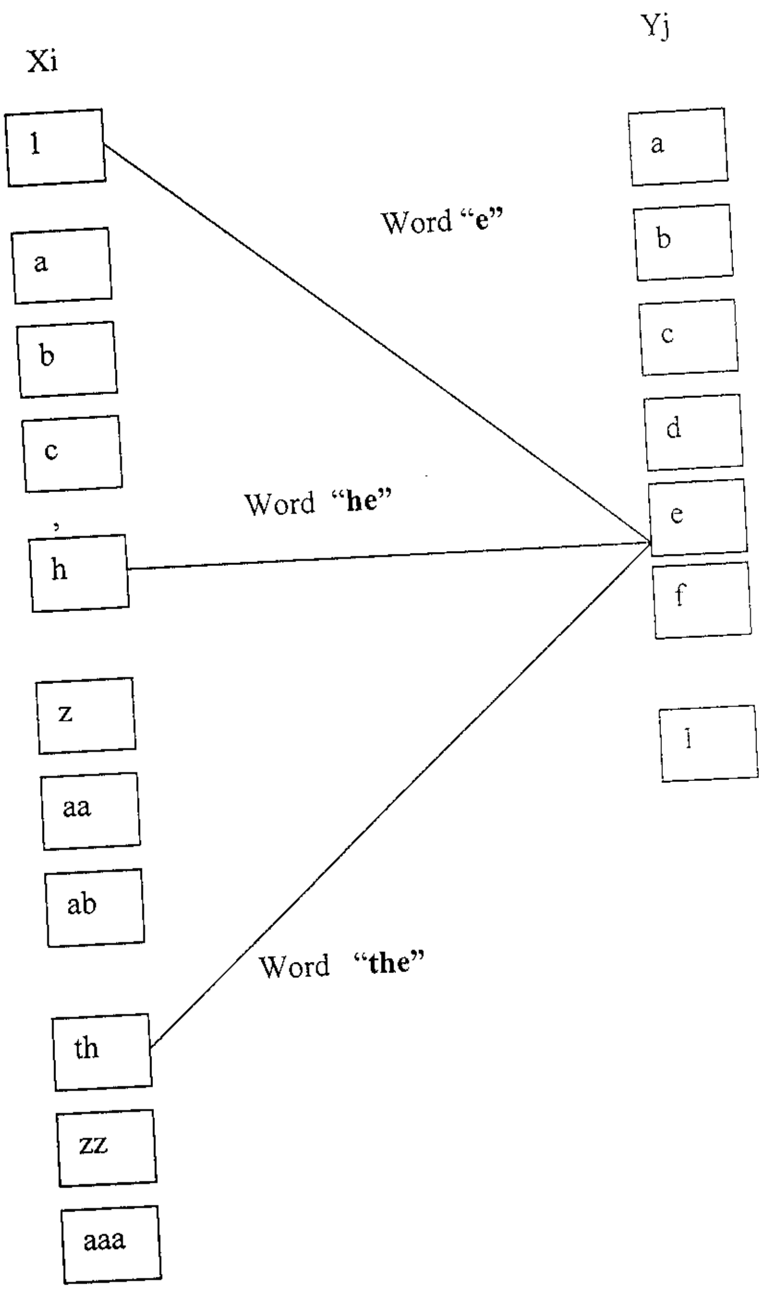
Neural network

It is predicting characters based on context. It is equivalent to a 2 layers network, with one input (x_i) for each possible context and one output (y_j) for each character in the alphabet (below). There is a weighted connection ($w_{i,j}$, not all of them are shown) between every input and every output.

To make a prediction, such as P(e|th) (that ***th*** will be followed by *e*), all of the contexts that are present (1, **h**, and **th**) are set to 1, and all others to 0. (The input labeled "1", representing the 0-order context, is always on). Then the outputs are calculated as follows.

$$y_j = g(\sum_i w_{i,j} x_i)$$

where $g(x) = 1/(1 + e^{-x})$



Then y_i is the probability that character *i* is next. For example, **P(e|th) = g(w_e + w_{he} + w_{the}).**

The g(x) function has a range (0, 1), so it always represents a valid probability. It is an increasing function. For instance:

- g(-4) = 0.018
- g(-2) = 0.119
- g(-1) = 0.269
- g(0) = 0.5
- g(1) = 0.731
- g(2) = 0.881
- g(4) = 0.982

The predictor is a 2 layer neural network with n inputs, X1...Xn and 1 output. The inputs can be arbitrary functions of the previous data, intended to be functions, which are likely to predict the next data bit, Y.The output is an estimate of P(Y) = 1/(1 + e^(-SUM(i) Wi * Xi)).The weights Wi are incrementally updated to minimize the prediction error, E = Y - P(Y). The weight update is:

Wi = Wi + (RS + RL/sigma2) * Xi * E

where RS and RL are short and long term learning rates, and

sigma2 = (C0+d)*(C1+d)/(C0+C1+2d)

is the variance of the training data in context Xi, where C0 and C1 are the counts of Y=0 and Y=1 in context Xi, and 0 < d <= 1 is a parameter to avoid division by 0 (which would infer P(Y) = 0 or 1). In practice the counts are halved when either exceeds 250 to avoid 8-bit overflow.

The parameter RS is typically 0 to 0.5, higher values for data that Changes statistics frequently (like text to binary). 0 is optimal for Stationary data (uniform statistics). RL should be between 1/m and 1 for m (= 6) inputs. Low values are good for correlated inputs, 1 is optimal for independent inputs.

The m inputs are divided into 6 sets for a total of 2^22 = 4,194,304 Inputs. Only one input from each set is active at any time. The active input within each set is selected as a hash function of the immediate context, 1-4 previous bytes or 1-2 previous words, plus the 0-7 bits of the partial current byte, plus the position (0-7) within the current byte. The context sets are:

- The last byte and current 0-7 bits with a leading 1, concatenated

  to a 16 bit number, i. (Xi = 1, and Xj = 0 for 0 <= j < 64K and j!=i).
- The last 2 bytes plus current bits and position, hashed to a 22-bit number.
- The last 3 bytes plus current bits and position, hashed to a 22-bit number.
- The last 4 bytes plus current bits and position, hashed to a 22-bit number.
- The current word (letters A-Z, a-z) plus current bits, hashed to 22 bits.
- The current and previous words, plus current bits, hashed to 22 bits.

The 5 hashed context indices share a common space of 2^22 - 2^16 inputs that do not overlap the first set of 2^16 inputs. The last two contexts, the selected input xi depends only on the letters in the words and not on any character between them. The parameters are hand tuned to d=. 05,RS=0.06,RL=0.38.

Arithmetic encoding

Start with a range from 0 to 1, and divide it up in proportion to the input probabilities from the predictor. For instance, common letters like *e* and *t* get larger slices than rare letters like *q* and z. Then the encoder peeks at the next letter, and selects the corresponding slice to be the new (smaller) range. This is repeated for each input character. For instance, the input ***the*** might be encoded as follows:

```
  0              .7  .8     1
+-------------------------------+
| a |b| c|d| e  | ... | t | ..... |
+-------------------------------+
        /                 \
       /                   \
  .7        .74   .76        .8
+-----------------------------------+
| a  ||||e||  h   | i|||| o |..|
+-----------------------------------+
        /                 \
       /                   \
  .74      .746       .752    .76
+-----------------------------------+
|   a      ||   e      || i |.. |   "the" = .75  (11 in binary)
+-----------------------------------+
```

The final output is a number, expressed as a binary fraction, from anywhere within the resulting range. In the example above, we could pick 0.75 (.11 in binary) because it is between .746 and .752. Thus, the steps to encode ***the*** are

1. Predictor assigns, say, P(a) = .07, P(b) = .01, ..., P(t) = .10, ..., P(z) = .001
2. Encoder sorts the input, assigns [0, .07] to *a*, [.07, .08] to *b*, ..., [.70, .80] to *t*, ...
3. Next input character is *t*, new range becomes [.70, .80]
4. Predictor (knowing the last letter was *t*) assigns P(a|t), P(b|t), ..., P(h|t) = .2, ... P(z|t)
5. Encoder assigns a sub range to each letter, including 20% to h: [.74, .76]
6. Next letter is *h*, new range becomes [.74, .76]
7. Predictor assigns P(a|th), P(b|th), ... P(e|th) = .3, ...
8. Encoder assigns [.746, .752] to ***the***
9. Input is *e*, new range is [.746, .752]
10. End of input, output a number in the resulting range, say .75 (11 in binary)

The decoder knows the final result, (.75 or 11 binary). Thus the steps for converting this back to *the* is

1. Predictor assigns exactly the same probabilities as before, P(a) = .07, P(b) = .01, ..., P(t) = .10, ..., P(z) = .001
2. Encoder sorts the input, assigns [0, .07] to *a*, [.07, .08] to *b*, ..., [.70, .80] to *t*, ...
3. Code is .75, so select *t*
4. Predictor (knowing the last letter was *t*) assigns P(a|t), P(b|t), ..., P(h|t) = .2, ... P(z|t)
5. Encoder assigns a sub range to each letter, including 20% to h: [.74, .76]
6. Code is .75, so next letter must be *h*
7. Predictor assigns P(a|th), P(b|th), ... P(e|th) = .3, ...
8. Encoder assigns [.746, .752] to *the*
9. Input is .75, so new range is [.746, .752] and output is *e*

Predictive arithmetic encoding is within 1 bit of optimal, assuming the probabilities are accurate. Shannon proved the best we can do to compress a string *x* is $\log_2 1/P(x)$ bits. In our example, we have P(the) = P(t)P(h|t)P(e|th) = 0.1 x 0.2 x 0.3 = 0.006, and $\log_2 1/0.006 = 7.38$ bits. We can always find an 8 bit code within any range of width 0.006 because these numbers occur every $1/2^8$, or about every 0.004.

4.3.1.6 Implementation

Neural network use 4 bytes of memory for input. The weight W_i is represented as a 16 bit signed number with a 10 bit fraction. The counts N(0) and N(1) are represented as 8 bit unsigned numbers with a 1 bit fraction To prevent over flow, both counts are divided by 2 whenever either count exceeds 250.

The arithmetic encoder stores the current range as a pair of unsigned 32 bit numbers, writing out the leading bytes whenever they match. Using bits shifts in the place of division when possible, using table lookup to compute g(.), and representing the weights as an array of structure.

Neural network looks at the last few characters and predict based on what happened before in the same context .For instance, if the last 2 characters **th** , then we can estimate the probability that the next letter is **'e'** is p(e/th)=N(the)/N(th),where N(x) is the count of the occurrence of x in the previous input. This works if the count are large, but if N(th)=0,and estimate $p(e/h)=N(he)/N(h)$.

IF the context is also novel, then we fall back to an order 0 context :$p(e)=N(e)/N$.

How weights are determined?

Initially all the weights are set to 0.Then after each prediction we look at the actual character and adjust all the weights in order to reduce the error. The adjustment has the form,

$$\mathbf{W}_{i,j} \leftarrow \mathbf{W}_{i,j} + v\mathbf{X}_i \mathbf{E}_j$$

Where $E_j = Y_j\ P(j)$ is the error, the difference between the actual and the expected next character, and the constant v is the learning rate.

What is learning rate?

This determines hoe fast the weights are adjusted. Usually this is set ad hoc manner. If it is too small, then the network will learn too slowly. If it is too large, then it might oscillate between very large positive and negative weights. Either way, the result is poor compression.

If only one input is active, then it can solve for the weights directly. If the desired output is p, then for weight between them,

$$W=g^{-1}(p) = \log p/(1-p)$$

If let $p=N(1)/N$, then it ca n get the same effect by setting the learning rate to $(N(0)+N(1)/N(0)N(1)$.This requires that it store along with each weight the counts N(0) and N(1),the number of times the output was 0 or 1 when the input was active .

The counts are initialized to 0.5 instead of 0 to avoid probabilities of 0 or 1.

Long tern learning rate:

The effect is to base the predictions on all the input, regardless of how long ago it occurred. If the input is

That thatch thaw the theatre theft th_

Then it will assign the same probability to a and e, even though the most recent statistics suggest that e is more likely.

Short term learning rate

The effect is to bias the statistics in favor of the last 1/NS occurrences of the context, and "forgetting" the earlier examples.

4.3.1.7 Algorithm

Encoding

// In compression mode, make the lower or upper sub range the new range according to y. In decompression mode, return 0 or 1 according to which sub range x I in, and make this the new range. X1 < = x < = x2 should be maintained as the last 4 bytes of compressed data :IN compressed mode, write the leading bytes of x2 that match x1. In decompression mode, shift out these bytes and shift an equal number of bytes into x from the archive. //

```
Ulong p = ~pr();
Ulong Xmid = x1; //x1+*p(x2-x1) multiply without overflow
Cons ulong Xdiff = x2-x1;
Calculating xmid from the probability:
{
   //split the range x^32 into 5 sets as xdiff >=  2^28,2^24,2^20,2^16 and xdiff < 2^16
     Xmid + =p*Xdiff;
 if(compress mode)
 {
   If  (the current bit(y) is 1)
 x1= Xmid+1;
   else
    x2 = Xmid;
 }
 else (decompress mode)

  if (last 4 bytes (x)<= xmid)
       {
        y=0;
        x2 = Xmid;
    }
else
{
  y = 1;
  x1 = Xmid + 1;
 }
 }
 Predictor. Update (y)   //predictor part
 while (MSB(8 bits) of x1 & x2   are same)
  {
  if(compress mode)
  {
     write the most 8 bits of x1(or) x2  into the archive file;
  }
  x1=x1<<8;
  x2=(x2<<8)+255;
  else(decompress mode)
   {
              c=read one character from the archive file;
              if (c==EOF)
              c=0;
              else
              ++ compressed data;
               x=(x<<8)+c;
    }
    x1=x1<<8;
    x2=(x1<<8)+255;
     }
    }
```

```
return y;
}
```

Destructor

```
if(compress)
{
   write the remaining encoded value
    into the archive file.
}
```

Constructer

```
if(decompress)
{
x = last 4 bytes from the archive}
```

Neural network

Assumptions

Number of layers is two.

Number of neurons are 2^22

Long term learning rate RL is 0.38.

Short term learning rate RS is 0.06.

Parameter to avoid division by zero   d is 0.5.

Number of contexts NX are 6.

C0- number of 0 bits present in that particular context.

C1- number of 1 bit present in that particular context.

Ulong S0-last 0-7 bits of current byte with leading 1   // to find the EOB (1-255).

Ulong S7, S3- last 7 complete bytes of input: 7-4,3-1.

Ulong SWO, SW!-hash of current and previous words.

Sigma sigma2- computes RL/sigma^2

S0=1,S3=S7=SW0=SW1=0:

Pr (32768):

Active set of contexts is set to zero.

Process

Train network to predict bit y next time, save next prediction in pr

```
Error=y-probability p ():
RS=0.06.
CMAX=250// to avoid 8-bit over flow
For I=1 to 5
{

 w=weight(active context + position);
 if (C0 of context>CMAX) Y
 C0= C0>>1;
 Else
 C0=C0+1-y;
 If(C1 of context >CMAX)
 C1=C1>>1;
 Else
 C1=C1+y;
 Weight updation:
 W + = error*(RS+sigma(c01));       //C01 is the concatenation of C0 and C1
 }
 S0 = (s0 << 1) /y    //appending the current bit (y) to S0.
 If (S0 >= 256) //EOB is reached
 {
   If (is alpha (S0 &255)
 {
   SW0 = SW0*997+S0
 Else
 {
      if (SW0)
      {
            SW1=SW0
             SW0=0;
      }
}
S7=(S7<<8) | (S3>>24); // Bytes 7-4 back
S3=(S3 | (S0 & 255))<<8; //last 3 full bytes, padded with a 0 byte
S0=0;
//select active inputs (hash x [1] t0 x [4] modulo primes <2^22-2^16-2^8)
x[0] = 4128768+(S3&0xffff);  //last byte
x[1] = (S3 & 0xffffff)%4128511; //last 2 byte
x[2] = S3%4128493; // last 3 bytes
x[3] = (S3+S7*0x3000000)%4128451;//last 4 bytes
x[4] = (SW1+SW0*29)%4128401: last 2 words
x[5] = SW0%4128409: //last word
}
//compute next p
{
summing the weights
sum+=w.w;
}
pr  =  g(sum);
}
```

4.1.1.8 Test Results

The program has been developed in c++ and tested with sample files. The results are shown in appendix –A.

4.1.1.9 Summary

Compression within 2% of best known, at similar speeds

50% better than *compress, zip, gzip*

Fast because

5. RESULTS OBTAINED

5.1 Test Results

The program has been developed in c++ and tested with sample files.

5.2 Summary

Compression within 2% of best known, at similar speeds

50% better than *compress, zip, gzip*

Fast because

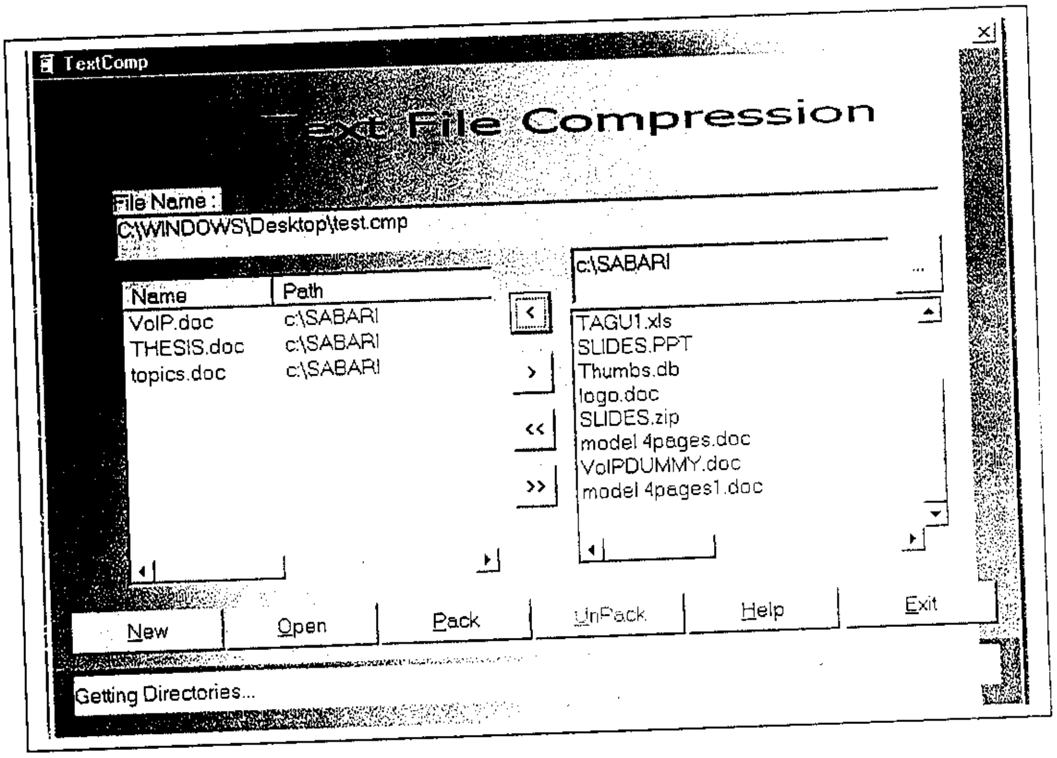
- Fixed representation - only output layer is trained

(6x faster)

- One pass training by variable learning rate (25x faster)
- Bit-level prediction (16x faster)
- Sparse input activation (6 of 4 million, 80x faster)

The figure shows the GUI output, which is highly user interface. The files to be compressed are placed in a new archive and the user provides the new archive file Name and places the archive in the newly specified path.

By choosing the directories option, user may choose the necessary files to be compressed by selecting and placing them in a new pane area.



Name	Path
VoIP.doc	c:\SABARI
THESIS.doc	c:\SABARI
topics.doc	c:\SABARI

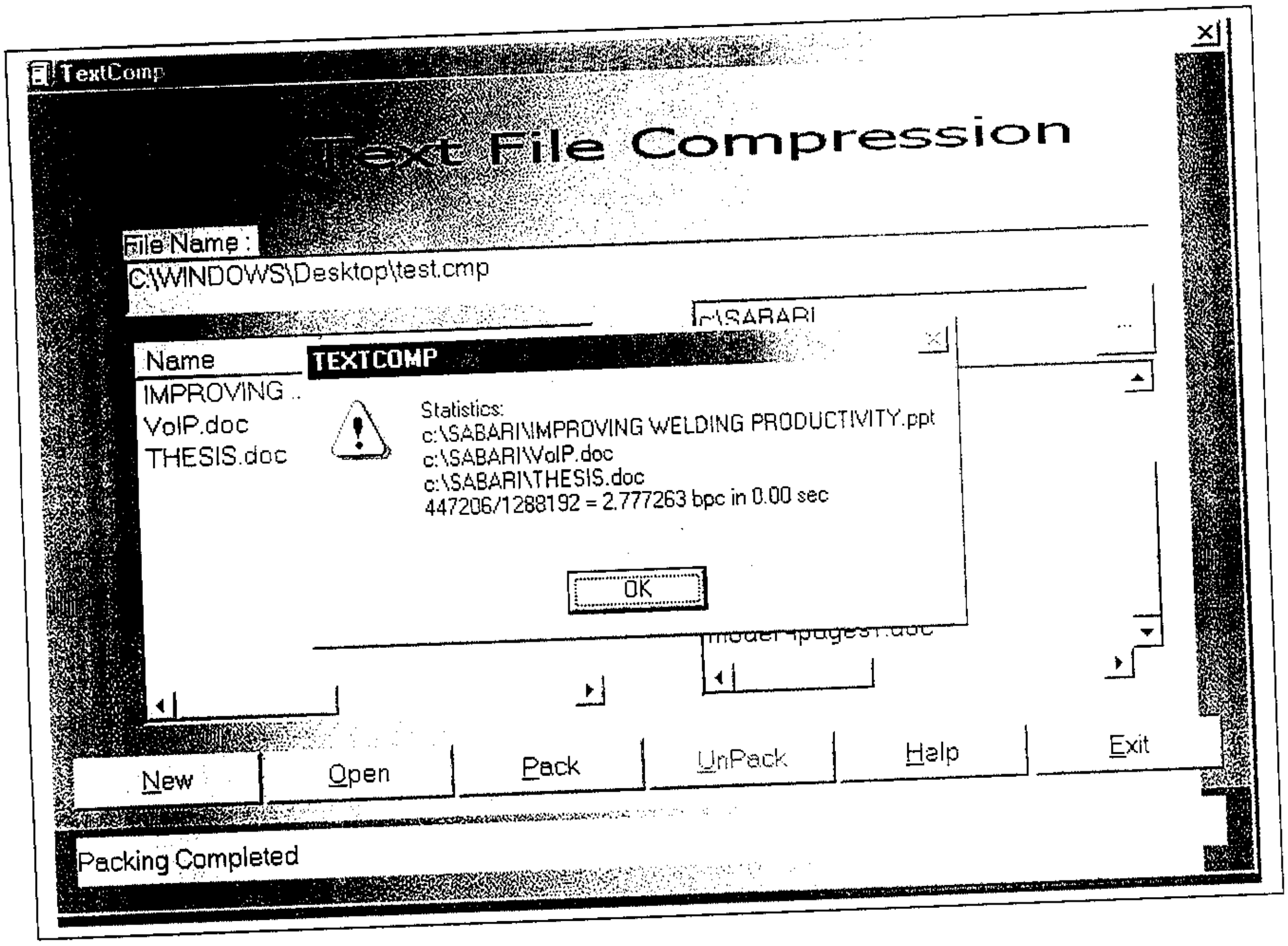
TAGU1.xls
SLIDES.PPT
Thumbs.db
logo.doc
SLIDES.zip
model 4pages.doc
VoIPDUMMY.doc
model 4pages1.doc

Getting Directories...

GUI FORMAT OF COMPRESSION / DECOMPRESSION PROGRAM

COMPRESSION

The following figure shows the information about the packed files and their corresponding path, where they have been taken from



C:\WINDOWS\Desktop\test.cmp

c:\SABARI

IMPROVING ..
VoIP.doc
THESIS.doc

TEXTCOMP

Statistics:
c:\SABARI\IMPROVING WELDING PRODUCTIVITY.ppt
c:\SABARI\VoIP.doc
c:\SABARI\THESIS.doc
447206/1288192 = 2.777263 bpc in 0.00 sec

OK

Packing Completed

PACKING TFILES

5. CONCLUSION AND FUTURE OUTLOOK

6.REFERENCES

16."Borland C++ in Depth" version 5 by WILLIUM H. MURRAY 111 & CHRIS H.PAPAS 1996- MCGRAW HILL

IEEE TRANSACTIONS

*. Schmidhuber, Jürgen, and Stefan Heil (1996), "Sequential neural text compression", IEEE Trans. on Neural Networks 7(1): 142-146.

*Ziv, Jacob and Lempel, Abraham, A universal Algorithm for sequential data compression", IEEE Transactions on Information theory IT--23 (1977),337-343.

*.Witten, Ian H., Timothy C. Bell (1991), "The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression", IEEE Trans. on Information Theory, 37(4): 1085-1094

Web sites

*.http;//www1.ics.uci.edu/~dan/pubs/DC-Sec.html , JUNE 26

*.http//www.cs.duke.edu/~natsev/thesis/node.html , JUNE 25

*.Buurrows, M., and Wheeler.D.J 1994 A Block-Sorting Lossless Data compression Algorithm. Digital systems research center.
http;//getekeeper.dec.com/pub/DEC/SRC/rearch-reports/abstracts/src-rr-124.html. , JULY 3

*.Gilchrist, Jeff (1999), Archive Comparison Test, http: // act.net/act.html , JULY 5

*.Arithmetic coding concept, http://www.arturocampus.com 22-jul-1999 , JULY [illegible]