Text Compression via Alphabet Re-Representation

(extended abstract)

Philip M. Long* Apostol I. Natsev† Jeffrey Scott Vitter†

Abstract

We consider re-representing the alphabet so that a representation of a character reflects its properties as a predictor of future text. This enables us to use an estimator from a restricted class to map contexts to predictions of upcoming characters. We describe an algorithm that uses this idea in conjunction with neural networks. The performance of this implementation is compared to other compression methods, such as UNIX compress, gzip, PPMC, and an alternative neural network approach.

1 Introduction

In this paper, we describe a new avenue for improving the compressibility of text. The main idea is that changing the representation of the alphabet may prove beneficial for various text processing tasks, including compression.

The current state-of-the-art methods for compression such as PPM [1, 3] generally work in two stages: first they try to estimate the probability distribution of the next character in the given context, and then they entropy-code it using the predicted probabilities. Since the second step (e.g., a statistical arithmetic coding) is provably optimal with respect to the guessed distribution, we focus our attention on the first step.

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 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.

For a given context length, PPM does not impose any *a priori* (i.e., before encoding starts) constraint, implicit or explicit, on the function from previous characters to the

*Department of Information Systems and Computer Science, National University of Singapore, Singapore 119260, Republic of Singapore, plong@iscs.nus.sg. Supported by National University of Singapore Academic Research Fund Grant RP960625. Some of the work reported in this paper was done while this author was at Duke University supported by ONR grant N00014–94–1–0938 and AFOSR grant F49620–92–J0515.

†Computer Science Department, Duke University, Durham, NC 27708, {natsev, jsv}@cs.duke.edu. Support was provided in part by Air Force Office of Scientific Research, Air Force Material Command, USAF, under grants F49620–92–J–0515 and F49620–94–1–0217, and by an associate membership of the third author in CESDIS.

probability distribution on the current character. A neural network biases itself toward smooth functions; thus convergence of statistics is faster if this bias turns out to be justified. Our re-representation is aimed at making this true.

2 Algorithm PSM

The proposed algorithm, which we have named PSM (Prediction by Smooth Mapping), is summarized below:

1. *Compute an alphabet re-representation (a pre-processing step done off-line).* This step doesn't have a match in PPM-like methods, and the purpose of its introduction here is to facilitate Step 2.

2. *Model the probability distribution of the next character given a certain context.* This step is similar to the probability modeling step of PPM methods, the only difference being that it is done by a neural network in an attempt to efficiently capture the smoothness promised by the previous step.

3. *Use a statistical (arithmetic) coder to get the final output stream.* This step is exactly the same as in PPM. Special care is taken when the neural network's prediction is very poor and the probability of the next character is practically zero. If that happens, a special escape symbol is transmitted and the next character is encoded with a uniform distribution.

We propose to construct the re-representation while taking into account its effect on the resulting compression algorithm. We achieve this by viewing the re-representation as a neural network layer (closest to the inputs) from contexts to probability distributions. By specifying the right error-criteria to the neural network we can make sure that we have optimized everything with respect to our primary goal—minimizing the number of output bits required to encode a file. In Section 4, we derive a training update rule that performs gradient ascent on the log-likelihood.

The first two steps are therefore combined into a single feed-forward multi-layer back-propagation neural network of a particular architecture (see Section 3 for the specifics). The first layer of weights corresponds to the re-representation, while the rest of the network corresponds to the probability modeling. The network is trained off-line over a large set of training data, and then the weights that correspond to the new re-representation are fixed. After training, we operate on the assumption that a good re-representation is already computed and we thus have a smooth mapping from the context domain to the probability distribution domain. In this sense, we can treat the first layer of the network as a re-representation layer that pre-processes the input (essentially via a table look-up) before it passes it on to the rest of the network. Since Step 1 is done only once—during training—we need to perform only Steps 2 and 3 when we actually process a file on-line.

3 Algorithm Implementation

Our network takes as an input the current context, and outputs a number between 0 and 1 for each letter in the alphabet. These numbers are normalized so that they add up to 1, and so the result is the probability distribution of the next character given the particular context. Then, a statistical coder, such as arithmetic coding, uses all the probabilities to encode the actual character that appears next, and the error of the network is propagated back to the lower levels (the output corresponding to the actual encoded character gets 1 as a target value to be propagated, and all other outputs receive a target of 0). Since

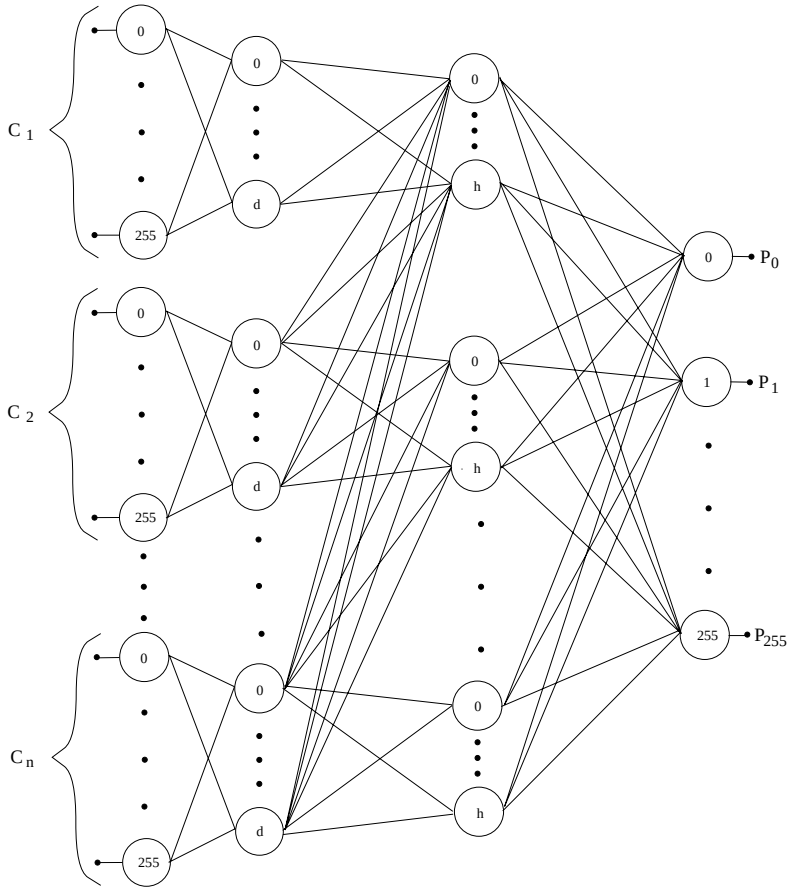


Figure 1: Architecture of the neural network demonstrating weight partitioning/blending. Parameters include: context size n, feature space dimensionality d, hidden chunk factor h.

the decoder has access to the same information the encoder uses for encoding (the context has already been decoded), it can decode the next character successfully, and update the weights of its equivalent neural net. Therefore, the network need not be transmitted.

Figure 1 illustrates the particular architecture of our neural network. The input layer consists of n groups of 256 input nodes each (the alphabet size can actually vary), where n is the context size. Each group of 256 nodes encodes exactly one character—the one that appears some k places back from the current point in the input string. For example, suppose we have the string *testing* as input, and we have just read the second t. If the context size of our model is three, the context will be est, and each of the characters e, s, and t will be encoded by a separate group of 256 input nodes. In the case of the letter e for example, all but the 101st node will be zero, the 101st node being 1 because the ASCII code of the character e is 101. The input consists of $256n$ nodes, all but n of which are 0.

Similarly to the input layer, the first hidden layer consists of n groups as well but each one has d nodes, where the parameter d denotes the dimensionality of the feature space. The weights between the first two layers correspond to the re-representation that we mentioned in previous sections. Note that only a single node will be non-zero in each group of 256 input nodes (namely the one corresponding to the ASCII code of the character encoded by that group). Therefore, the activation values in the kth group of nodes in the first hidden layer actually corresponds to the d-dimensional embedding of the character that appears

k places in the text before the character that we are trying to predict. Every character in the context is thus mapped to a d-dimensional vector, where each coefficient reflects the extent to which that character has a certain hidden feature. Note also that the characters have different representations according to how far away they are from the character to be predicted. Thus, this architecture allows us to capture the different statistical properties of characters as predictors of characters different distances further in the file. After the network is trained, all weights (i.e., character embeddings) between the first two layers are fixed so that retrieving a character's re-representation is essentially done via a look-up table.

The rest of the network is similar to traditional 3-layer networks. The presence of an additional hidden layer is warranted by the necessity to be able to learn non-linearly-separable functions, and the lack of further layers is motivated by our desire to restrict the network size so that it can generalize better. The second hidden layer, then, consists again of n groups, each containing h nodes and corresponding to a letter in the context. This time, however, the separate groups are not mutually independent but are rather combined in a blending fashion so that the ith group is fully interconnected with all but the first $i-1$ groups in the previous layer. This architecture is motivated by the ability to separate the predictions that are generated by a higher order context model from those generated by a lower order context model. This way, the nodes in the ith group of nodes in the second hidden layer have the full information of an order-i model that uses contexts of size up to i. The final probabilities are combined in the output layer so as to allow the network to weigh the different models in a different way. The parameter h simply reflects the complexity required to capture the important characteristics of an order-k model. This architecture resembles other statistical approaches such as PPM with blending, where the different models give separate estimates, which are then combined for the final prediction in a weighted blending fashion. Finally, the output layer, as we mentioned before, consists of 256 nodes, each corresponding to the probability that its character will be the next one in the string. The size of the output layer is fixed to 256 nodes.

4 Neural Network Update

In this section we consider the design of our network and the derivation of the neural network update for our application. We use a 4-layer feed-forward backpropagation neural network with a sigmoidal activation function for all hidden nodes, and a normalized exponential activation function for the outputs. The cost function is selected to maximize the log-likelihood of the data given the network, and we argue that our specific choice of a cost function is optimal for compression purposes.

4.1 Background on design issues

In this section we address some issues related to the design of neural networks (cf. [2]). Let $\mathcal{D} = \langle \vec{x}_i, \vec{t}_i \rangle$ denote the observed data where $\vec{x}_i$ is the ith input vector (i.e., the context), and $\vec{t}_i$ is the target vector (i.e., the character which appeared next). Let $\mathcal{N}$ be the neural network that is designed to learn $\mathcal{D}$. The usual Bayesian motivation leads us to maximize

$$\ln P(\mathcal{D} \mid \mathcal{N}) = \ln \left(\prod_i P(\langle \vec{x}_i, \vec{t}_i \rangle \mid \mathcal{N}) \right) = \sum_i \ln P(\vec{t}_i \mid \vec{x}_i \wedge \mathcal{N}) + \sum_i \ln P(\vec{x}_i). \qquad (1)$$

Each $P(\vec{x}_i)$ does not depend on the network choice and can therefore be disregarded for our purposes. At this point, in order to get an expression for an optimal cost function we need to make some assumption about the type of the probability distribution $P(\vec{t}_i \mid \vec{x}_i \wedge \mathcal{N})$.

In [2] Rumelhart, Durbin, Golden, and Chauvin consider the general family of distributions

$$P(\vec{t} \mid \vec{x} \wedge \mathcal{N}) = \exp\left(\sum_i \frac{(t_i\theta - B(\theta)) + C(\vec{t}\phi)}{A(\phi)}\right), \tag{2}$$

where θ is related to the mean of the distribution, ϕ is the overall variance, and the functions $A(\,)$, $B(\,)$, and $C(\,)$ are specified individually for each member of the family of distributions. As it turns out, for any such distribution, we can derive a cost function $\mathcal{E}$ that maximizes the log-likelihood term given by equation (1) so that its gradient with respect to the net input at output node j is given by $\frac{\partial \mathcal{E}}{\partial \mathrm{net}_j} \propto \frac{t_j - a_j}{\mathrm{var}(a_j)}$. We can further choose an activation function for the output nodes that cancels the variance term in the above expression so that the gradient is always proportional only to the difference between the target values and the actual output values of the network. The multinomial distribution is a special case of the above family. The corresponding energy function is given by $\mathcal{E}_{multinomial} = \sum_i \sum_j t_{ij} \ln a_{ij}$, where index i ranges over the observations, and index j ranges over the output nodes. The most appropriate activation function for the multinomial case is the normalized exponential function. In [2], the authors argued that the multinomial case is most suitable when the network is supposed to make 1-out-of-n classification. In that case the output is treated as a probability distribution, and the ith output node corresponds to the probability that the pattern goes to the ith class. Since this is exactly the case in data compression applications (we are trying to predict one letter from an alphabet of fixed size), for our purposes we use the energy function given above along with the normalized exponential activation function for the output nodes: $a_j = e^{\mathrm{net}_j} / \sum_k e^{\mathrm{net}_k}$.

If we interpret the desired and the actual outputs as probability distributions over the fixed alphabet, then the multinomial energy term is exactly the opposite of the number of bits we would spend to encode the input string. In other words, if character α_j has a probability t_{ij} of occurring at the ith position in the text, and the neural network's estimate of that true probability is a_{ij}, then the expected number of bits that need to be transmitted to uniquely encode the ith character is equal to $\sum_j t_{ij} \log \frac{1}{a_{ij}} = -\sum_j t_{ij} \log a_{ij}$. When we sum over the position i in the text, we obtain the total number of bits needed to encode the input string. As we can see, the expression for the energy function we selected describes exactly that quantity, up to a constant factor of $-\frac{1}{\ln 2}$. Therefore, by maximizing the chosen energy function, we are not only maximizing the log-likelihood of the data given the network but we are also directly minimizing the total number of bits required to encode the data.

4.2 Derivation

Having chosen the architecture, the cost function, and the activation functions we move on to derive the particular formulas used for updating the weights of the network during training. While the general backpropagation model can be found in standard references such as [4], we are not aware of references that derive it in the context of normalized exponential outputs and entropy-like cost functions. Therefore, we include the full derivation here, differing from the standard one mainly in equations (6)–(8). We use gradient descent, setting

$$w_{ij}^{new} = w_{ij} + \eta \frac{\partial \mathcal{E}}{\partial w_{ij}} = w_{ij} + \eta \frac{\partial \mathcal{E}}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ij}} = w_{ij} + \eta a_i \delta_j, \tag{3}$$

where w_{ij} is the weight between nodes i and j, η is the learning rate, $\mathrm{net}_j = \sum_k a_k w_{kj} + b_j$ is the net input and b_j is the bias at the jth node (the index k ranges over all the nodes in the previous layer), and δ_j denotes the gradient with respect to the net input at node j.

If the jth node is a hidden unit, we express δ_j as follows:

$$\delta_j = \frac{\partial \mathcal{E}}{\partial \mathrm{net}_j} = \frac{\partial \mathcal{E}}{\partial a_j}\frac{\partial a_j}{\partial \mathrm{net}_j} = \frac{\partial a_j}{\partial \mathrm{net}_j}\sum_k \frac{\partial \mathcal{E}}{\partial \mathrm{net}_k}\frac{\partial \mathrm{net}_k}{\partial a_j} = S_j^{'}(\mathrm{net}_j)\sum_k \delta_k w_{jk}, \quad (4)$$

where $S_j^{'}(\mathrm{net}_j)$ denotes the derivative of the sigmoid activation function of the jth node evaluated at the net input to the node, and the index k ranges over the nodes in the next layer. This expression of δ_j for hidden nodes in terms of the precomputed δ_k values of the nodes in the next layer forces us to work backwards. We first calculate the gradients at the output nodes, and then backpropagate the error to the hidden nodes. Now, for the derivative of the sigmoid function, we obtain the following expression:

$$S_j^{'}(\mathrm{net}_j) = \left(\frac{1}{1+e^{-\mathrm{net}_j}}\right)^{'} = \frac{1}{1+e^{-\mathrm{net}_j}} - \left(\frac{1}{1+e^{-\mathrm{net}_j}}\right)^2 = a_j - {a_j}^2 = a_j(1-a_j), \quad (5)$$

leading to the following final form for a hidden node's gradient: $\delta_j = a_j(1-a_j)\sum_k \delta_k w_{jk}$.

As we know from [2] our particular choice of an output activation function for our cost function results in $\delta_j = t_j - a_j$ as the gradient at the jth output node. For completeness purposes, however, we shall include the full derivation of that fact. The formula for $\mathcal{E}_{multinomial}$) tells us that the cost function is a summation of independent cost terms for each observation. Theoretically, then, to obtain the true gradient we would need to consider the cost over all observations, and then move in the optimal direction. However, for convenience, at each iteration we shall consider the cost of that iteration only in order to obtain the gradient. This modification does not introduce a large error provided that we use a sufficiently small learning rate, and take small steps in the direction of the gradient after each observation. The cost function would then be $\mathcal{E} = \sum_l t_l \ln a_l$, where l ranges over the output nodes; t_l and a_l are the target and the actual activation values, respectively, of node l. Note that for our purposes all t_l terms will be 0 with a single exception, which will be equal to 1. We can therefore notice that the cost function will actually be equal to $\ln a_{ch}$, where ch is the index of the next character in the text. Again, it can be seen that the cost function is the inverse of the number of bits one would spend to encode the character ch (that number would be equal to $-\log a_{ch} = -\frac{\mathcal{E}}{\ln 2}$). We could take advantage of that special form of our cost function to simplify things a little but we prefer to give the derivation in the case of a general probability distribution target vector because it will work in other applications as well. Thus, for output nodes we have $\delta_j = \frac{\partial \mathcal{E}}{\partial \mathrm{net}_j} = \sum_k \frac{\partial \mathcal{E}}{\partial a_k}\frac{\partial a_k}{\partial \mathrm{net}_j}$. Furthermore,

$$\frac{\partial \mathcal{E}}{\partial a_k} = \frac{\partial\left(\sum_l t_l \ln a_l\right)}{\partial a_k} = \frac{\partial\left(t_k \ln a_k\right)}{\partial a_k} = \frac{t_k}{a_k}, \quad (6)$$

$$\begin{aligned}
\frac{\partial a_k}{\partial \mathrm{net}_j} &= \frac{\partial\left(\frac{e^{\mathrm{net}_k}}{\sum_l e^{\mathrm{net}_l}}\right)}{\partial \mathrm{net}_j} \\
&= \frac{\frac{\partial e^{\mathrm{net}_k}}{\partial \mathrm{net}_j}\sum_l e^{\mathrm{net}_l} - e^{\mathrm{net}_k}\frac{\partial\left(\sum_l e^{\mathrm{net}_l}\right)}{\partial net^{\mathrm{net}_j}}}{\left(\sum_l e^{\mathrm{net}_l}\right)^2} \\
&= \frac{[j=k]e^{\mathrm{net}_k}\sum_l e^{\mathrm{net}_l} - e^{\mathrm{net}_k}e^{\mathrm{net}_j}}{\left(\sum_l e^{\mathrm{net}_l}\right)^2} \\
&= \frac{e^{\mathrm{net}_k}}{\sum_l e^{\mathrm{net}_l}}\,\frac{[j=k]\sum_l e^{\mathrm{net}_l} - e^{\mathrm{net}_j}}{\sum_l e^{\mathrm{net}_l}} \\
&= a_k\left([j=k] - a_j\right), \quad (7)
\end{aligned}$$

where the Boolean expression $[j = k]$ evaluates to 1 if j is equal to k, and to 0 otherwise. Now, plugging formulas (6) and (7) into the gradient term for output nodes we get

$$\delta_j = \sum_k \frac{t_k}{a_k} a_k ([j = k] - a_j) = \sum_k t_k [k = j] - \sum_k t_k a_j = t_j - a_j \sum_k t_k = t_j - a_j. \quad (8)$$

The last equality follows from the fact that the t_k terms make up a probability distribution and therefore add up to 1. We have thus derived the formulas for the gradient with respect to the net input of both output nodes and hidden nodes. After computing all the delta-terms, we use equation (3) to update all the weights. This concludes the derivation of the neural network model.

5 Setting Parameters

First of all, the learning rate used in the gradient descent weight-update is crucial during the training phase. Furthermore, since in data compression applications the amount of training data for the network is relatively large, our main factor in determining the learning rate was not the optimal rate of convergence. This decision is motivated by the fact that, even with a small learning rate, the network will eventually learn its function provided that it is trained sufficiently long. Our main concern then was to make sure that the learning rate is not too big so as to result in oscillation on the gradient curve after extensive training. We have therefore used a decaying learning rate which was inversely proportional to the square root of the iteration number. This is a somewhat standard practice that semi-automates the process of learning rate adjustment, and we have adopted it for our application as well.

The other crucial parameters that affect the performance of the network are the ones that control its size, namely, the context length n, the feature space dimensionality d, and the "hidden chunk factor" h. For the context size we have used the values 5 and 10 because studies have shown that contexts of size between those numbers are the ones most heavily used for prediction purposes. The other two parameters d and h were estimated adaptively by monitoring the performance of the network on the training set and increasing the values of the two parameters whenever it appeared that the network converged to some local minimum (i.e., the improvement over the past several iterations was not significant). On the one hand, values that are too small will result with a network that is not big enough to learn the function we are modeling. On the other hand, values that are too large will prompt too many degrees of freedom which will probably improve the learning step during training but will certainly harm the generalization capabilities of the network (i.e., there will be overfitting). Therefore, estimating the right network size is of paramount importance for such applications. The neural net community has proposed several strategies for dealing with the overfitting problem, and they are mainly of two types:

- learn conservatively, and expand the size only if you have to.
- expand freely until you get the best fit on the training data, and then prune the network to improve generalization.

We have adopted the first approach for dealing with overfitting: first, by imposing *a priori* structure on the network through blending and partitioning; and second, by doing adaptive size adjustments through manipulating the d and h parameters during training.

6 Results and Discussion

We have run two sets of experiments to test our approach. The first set is identical to the experiments reported in [5], where the authors propose a similar neural network approach

that consists of the same input/output structure but uses only one hidden layer. As reported in [5], the number of hidden nodes used in their single hidden layer is 440, the context size is 5, and the alphabet size is 80. The matching configuration that we used had an alphabet size of 256 so that it can handle even binary files but all characters that were predicted with essentially a zero probability were combined together and treated as a single escape character. This way, the enlarged alphabet size does not hurt prediction but allows generality at the cost of a somewhat higher complexity. We used the same context size of five symbols, and the other parameters were eventually set to $d = 50$, and $h = 30$ (they were estimated adaptively during the course of training). Thus, the parameter choice resulted in a network with 1280 input nodes, 250 nodes in the first hidden layer, 150 nodes in the second, and finally 256 output nodes. However, since the weights between the first two layers correspond to the re-representation and are fixed during on-line file processing (i.e., they are retrieved as a table look up), the network used for actual compression was essentially a three-layer network with 250 input nodes, 150 hidden nodes, and 256 output nodes. The training time for PSM was very slow, though the complexity of the network is lower than that of the alternative neural network approach [5]. The learning rate used in the experiments reported in [5] was fixed to 0.2 but for our approach we used a decaying learning rate which started off at 0.2. The training set consisted of 40 articles from the German newspaper *Munchner Merkur*. Test set 1 consisted of additional 20 articles from the same newspaper, and test set 2 consisted of 10 articles from a different newspaper, *Frankenpost*, on which the networks were not trained in advance. All of the files in both the training and the test sets were smaller than 20 kilobytes. The results of the experiment are shown in Table 1. As can be seen from the results, our approach outperforms all other competitors, including the state-of-the-art PPMC text compression algorithm. The compression improvement ranges anywhere from about 15% (for PPMC and the other neural network approach) to more than 50% (for pack and compress). Compress and gzip use Ziv-Lempel algorithms [7, 8, 6], and are often used in practice due in part to their computational efficiency.

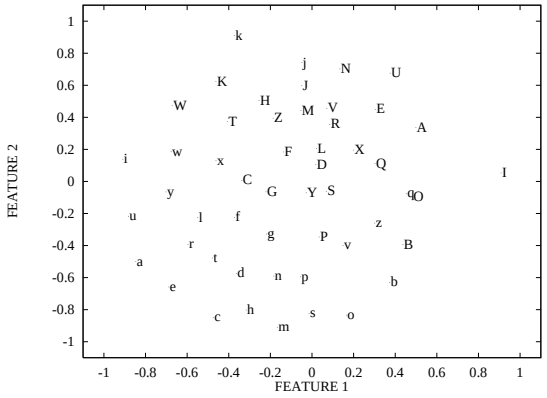
Method's Name	Average Compression Ratio (Variance)		
	Munchner Merkur	*Frankenpost*	*Jack London*
UNIX pack	1.74 (0.0002)	1.67 (0.0003)	1.78 (0.0001)
UNIX compress	1.99 (0.0014)	1.71 (0.0036)	2.45 (0.0060)
UNIX gzip -9	2.30 (0.0033)	2.05 (0.0097)	2.64 (0.0049)
PPMC method	2.70 (0.0069)	2.27 (0.0131)	3.54 (0.0984)
NN method in [5]	2.72 (0.0234)	2.20 (0.0112)	—
PSM method	**3.09** (0.0142)	**2.61** (0.0047)	**3.56** (0.0083)

Table 1: Compression performance of various methods on three test sets consisting of newspaper articles from *Munchner Merkur* and *Frankenpost*, and of books by *Jack London*.

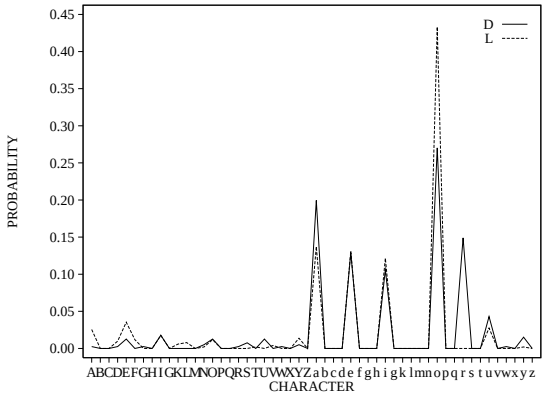
Since all the files in the previous experiment were relatively short (i.e., < 20 kilobytes), we designed a second experiment which uses longer files. While the purpose of the first experiment was to compare our approach with the alternative neural net approach proposed in [5], in the second experiment we wanted to primarily test our method against PPMC, which was at a disadvantage with the small files used in experiment 1. As we expected, PPMC performed competitively on the longer files, yielding compression ratios of over 3.5. The training set consisted of three books by Jack London totaling a little over 1 megabyte (*Sea Wolf, White Fang,* and *Call Of The Wild*), and the test set consisted of three other books (*Son Of The Wolf, Iron Heel,* and *People Of The Abyss*) by the same same author of approximately the same size. Our experiments with a context size 5 failed to outperform PPMC on these particular data sets, yielding compression ratios short of 3.5. However,

when we used a context length 10 and allowed the network to grow quite large (the size was adaptively adjusted to the final parameter estimates of $d = 45$ and $h = 40$), the performance of the network after 100 training iterations (with an initial learning rate of 0.3) actually surpassed that of PPMC. The results from the second experiment are also given in Table 1.

In order to interpret the results in terms of the alphabet re-representation, we have provided some plots of a particular re-representation obtained from training on the Jack London book set. We have used the weights in between the first two layers that correspond to the re-representation as described in Section 3, and we have shown only the 52 lower case and upper case characters from the English alphabet. The original dimensionality was 25 (that is, each character was mapped to a vector with 25 features). For visualization purposes, however, we have shown only a 2-dimensional embedding which preserves the original "distances" between pairs of characters as closely as possible. The algorithm used to compute the embeddings, given a distance matrix, operates in the following way: it positions an imaginary spring between each pair of characters so that the natural length of the spring is proportional to the desired distance between the two objects. Thus, if the objects are closer than desired, the spring will push them apart, and if they are too far apart, the spring will pull them together. After all springs are in place, the system is let to relax to an equilibrium state. This way, the algorithm preserves the similarity properties of the objects regardless of the new dimensionality of the feature space. Figure 2(a) shows the two-dimensional embedding of an order-1 re-representation (that is, only one character is used as a context for predicting the next character). One can clearly see the separation between lower-case and upper-case letters, as well as between vowels and consonants. We would also like to point out again that the original re-representation has 25 features, rather than 2, and is therefore much more expressive than can be visualized. Other features that may not be so intuitive should also be present but are harder to identify and explain.



(a) Two-dimensional embedding of an order-1 representation. Original dimensionality is 25.



(b) Probability distribution of the next character given the contexts D and L.

Another way to measure the success of the computed re-representation is to compare the probability distributions of the next character under two contexts that are mapped very close to each other by the new re-representation. To illustrate, we computed the distances between all pairs of characters under the new re-representation, and for each character we considered only its closest character. In other words, we picked the closest pairs of characters, and looked at the top few such pairs that corresponded to the smallest distances. Figure 2(b) superimposes the two distributions of the next character given an order-1 context of the letters D and L (the Euclidean distance between the 25-dimensional

embeddings of the two characters is 0.8512). From that graph we can easily see that the two distributions are similar in that they share the same peaks, as well as similar probabilities at those peaks. This means that the two contexts tend to isolate and predict the same few characters with high probabilities, and so we have a good reason to believe that the re-representation has been quite successful in accomplishing its task. This, by itself, is an achievement that can be used in other applications for purposes other than text compression.

7 Conclusion and Future Work

The main contribution of this project is the introduction of a new re-representation approach to the probability modeling step in data compression systems. The proposed technique is an initial attempt to gather semantic information about the geometric structure of text and to use it intelligently through the locality principle and neural networks.

The idea of imposing structure through re-representation proves advantageous in two important counts: it reduces time and space algorithm complexity (compared to the traditional neural network approaches), and at the same time it facilitates learning and generalization, thus providing better compression performance at a smaller cost.

Neural network algorithms are well known to be sensitive to parameters governing the learning process. We have not gone to great lengths to optimize our neural network training algorithm. The results obtained from our original attempts, together with the qualitative information represented in Figures 2(a) and 2(b), are encouraging. Future work may include a more thorough examination of some alternatives for the network's architecture, as well as a further study of other settings of the parameters (including longer than order-10 contexts).

In addition, the proposed approach of alphabet re-representation may prove useful for a variety of other problems. For instance, training the network on certain (known) types of text and using it to test the compressibility of other texts of unknown origin may give some insights about the texts' sources.

References

[1] Bell, T., Cleary, J.G., and Witten, I.H. *Text Compression.* Prentice Hall, 1990.

[2] Chauvin, Y., and Rumelhart, D. E. *Backpropagation: Theory, Architectures, and Applications.* Lawrence Erlbaum Associates, Inc., 1995.

[3] Cleary, J.G., and Witten, I.H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communication* 32 (1984), 396–402.

[4] Rumelhart, D., Hinton, D., and Williamson, R. *Parallel Distributed Processing, Explorations in the Microstructure of Cognition, Vol. 1: Foundations.* MIT Press, 1986.

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

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

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

[8] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24 (1978), 530–536.