DEEP LEARNING FOR RELIABLE STORAGE

A Thesis

by

ISHAN DHANANJAY KHURJEKAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Tie Liu
Co-Chair of Committee,	Anxiao Jiang
Committee Members,	Krishna Narayanan
	Alex Sprintson
Head of Department,	Miroslav Begovic

August 2018

Major Subject: Electrical Engineering

ABSTRACT

With the exponential increase of cloud based storage systems, it has become critical to reliably store data. Traditionally, methods for error correction have relied on duplication of data / introduction of artificial redundancy. Here, we leverage the natural redundancy present in the data using deep learning based techniques. Deep learning is a subset of machine learning algorithms that have given excellent results on a variety of tasks.

We describe DNN (deep neural net based) models for learning decompression in texts compressed by Huffman coding. Firstly, we work with noiseless texts following which we work with noisy texts. Next, we outline a model for bit erasure correction. For this, we present a DNN based model for bit erasure correction in uncompressed, ASCII encoded texts. Finally, we describe a model that does bit erasure correction for Huffman code compressed texts. Such an end-to-end system can be useful for cases when the codebook / encoding algorithm is not available and decoding / error correction needs to be done.

DEDICATION

To the highs and lows of research for they mirror life in more ways than one.

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude towards Dr. Anxiao Jiang for patiently guiding me throughout the thesis. His way of advising is unique and creates an atmosphere conducive to research. Apart from having deep understanding of the subject and innovative research ideas, Dr Jiang creates a personal bond with his students. His research legacy is worthy of emulation.

I would like to thank my parents for their unwavering support as well as for teaching me the importance of perseverance and humility early on in life. I would like to thank my committee members, Dr Tie Liu, Dr Krishna Narayanan and Dr Alex Sprintson for their guidance and support throughout the thesis work. I would like to thank my lab mates for the intriguing discussions about research and beyond.

I would also like to thank my extended family and of course, importantly, my invaluable friend circle for their constant love and support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by Professor Anxiao Jiang [co-advisor] of the CSE Department at Texas A&M Unviersity

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was not supported via any funding sources.

NOMENCLATURE

ECC	Error Correcting Code
LRC	Locally Recoverable Code
BCH	Bose Chaudhuri Hocquenghem code
ASCII	American Standard Code for Information Interchange
DNN	Deep Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory network
GPU	Graphics Processing Unit
API	Application Programming Interface
EOS	End of Sequence token
SVM	Support Vector Machine
MB	Megabytes
1D	one dimensional

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

In coding theory, one strategy to reliably transmit data over noisy channels is to employ error correcting codes (ECC). Error control consists of two parts: error detection and error correction. The idea behind error control is to introduce redundancy in the transmitted data by using an ECC. The encoded data after having being sent over the noisy channel, is then decoded. The error correcting codes have certain bounds on how many errors they can correct. Popular error correcting codes include Reed-Solomon codes, low-density parity check (LDPC) codes. Another class of codes to deal with cloud-based systems are the locally recoverable codes (LRC) [1]. LRC's provide availability along with reliability.

Simultaneously in the computing world, the field of deep learning has had major breakthroughs and has seen constantly increasing influx of researchers in the recent years. Deep learning refers to a subset of machine learning methods that focus on learning feature representations from input data in a supervised , semi or unsupervised manner [2]. Deep learning methods employ a stack of non-linear processing units. Majority of deep learning models are made up of multi-layer neural networks.

Researchers have recently been exploring the usage of deep learning models for error correction tasks. One approach uses deep learning methods for improving performance of short BCH codes [3]. Deep neural network based models for polar codes are explored in [4]. The recent direction and scope for exciting developments in error correcting codes using deep neural networks has prompted us to explore this research direction further.

Our research topic focuses on using deep learning techniques for bit level language inputs. Consider the case where a text file is stored in the binary format on a computing system. When huge amounts of data are stored, certain bits might get erased. Traditionally, to deal with storage issues, techniques like forward error-correcting codes have been used. Popular examples for error control, as mentioned above, include LDPC, Reed-Solomon etc. Our research aims to answer the question

that whether we can harness the power of deep neural network architectures for language storage. Here we work with text data. Simplistically put, we want the the neural network architecture to do the decoding (error correction) part.

For our research on language storage, we borrow ideas from a topic known as natural language processing. This field draws ideas from linguistics, computer science, probability and statistics to interpret and make inferences from language data. Researchers have achieved excellent results on a variety of natural language processing tasks using deep learning[5]. Examples include categorizing words in a sentence into different categories i.e named entity recognition [6], translating sentences from one language to another i.e neural machine translation [7] etc.

2. RELATED WORK

Researchers have started exploring novel error correcting methods using deep neural network based architectures. Nachmani et al, have used deep learning methods for improving channel decoding of linear codes [8]. The paper suggests replacing the belief propagation decoder with a recurrent neural network based decoder. In [3], an algorithm that shows improvements over the performance of belief propagation algorithm in high density parity check codes is proposed.

Gruber et al in [4], present a deep learning based decoder for both random and structured codes. This work is based on very short blocklengths ≤ 64 bits. The decoder is able to generalize for structured codes as well. This gives hope to further research in deep learning based decoder architectures. The results in the paper hint towards the fact the structured codes are easier to learn than random codes.

The problem of joint source channel coding has been extended to structured data like text in [9]. Traditionally, text data is encoded at source and then channel coding is performed for reliable transmission over noisy channel. The novelty in the authors work here lies in the fact that they use a deep neural network architecture for both the encoder and decoder. The paper uses a human judge for quantifying the performance of their methods at the word level.

Our research draws motivation from the initial research for deep learning encoder-decoder models. We also seek to leverage the natural redundancy present in text inputs by using deep learning methods for decoding. Previously, research has been done on exploiting natural redundancy to correct errors (in conjunction with ECC's) [10] and [11]. The authors propose natural redundancy (NR) based decoders for erasure correction in language as well as image data. Later, they combine the NR based decoders with LDPC codes.

In [12], the authors propose a method for polar decoding to exploit the natural redundancy present in language based sources. As is evidenced by the prior research in this direction, re-

searchers have looked to build connections between concepts from deep learning / machine learning and coding theory / information theory. In our research, we look to move further in this direction by using deep learning architectures for reliable storage systems. As the field warrants, we seek to quantify the performance of our algorithms in coding-theoretic performance measures.

3. USING DEEP LEARNING FOR LANGUAGE STORAGE

3.1 Why should we study reliable language storage ?

Text inputs typically exhibit complex features due to the rich variety possible in language data. When we have to deal with huge amounts of sensitive language data in storage systems, the reliability of data becomes critical.

From the point of view of a human mind, language consists of sentences or more fundamentally, characters. On the other hand, typically in computing systems, all data (text, image, video) is stored in the binary format. This motivates us to look at data in its most raw format: binary, rather than at the word / sentence level. Our research seeks to answer the question that whether we can build reliable language storage systems using deep neural network based architectures. Such research can help the academic community to make advances in designing effective algorithms for data encoding - decoding purposes. By working with binary data, we can work towards a hardware implementation of our error control architecture in the future.

3.1.1 Learning to decompress

The first part of our research deals with learning how to do decompression of texts compressed by Huffman code. The input to the system is a binary sequence of the compressed text and the output is the decompressed character sequence. Firstly, we use noiseless texts (ideal case) for our research. Following this, we generate noisy texts (general case) and obtain results.

3.1.2 Learning to correct erasures

This part of our research deals with learning to correct erasures in texts (both compressed and uncompressed). The input to this system, is a binary sequence with erasures and the output is a

prediction for those erased bits. First, we work with uncompressed texts encoded by ASCII code. Following this, we work with texts compressed by Huffman code.

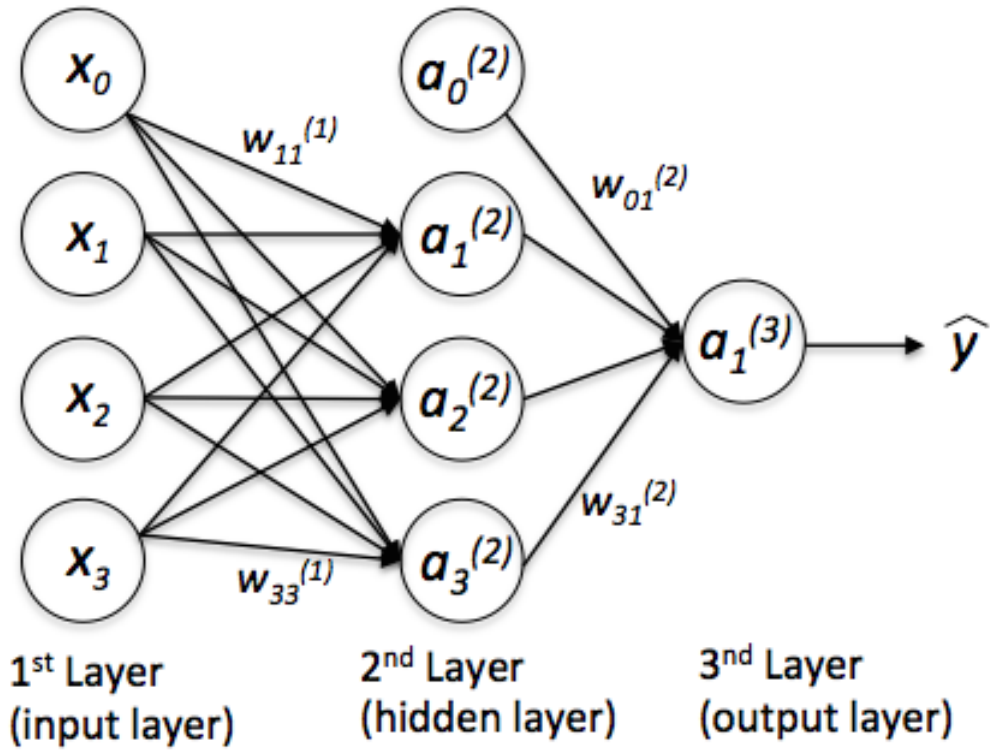

Figure 3.1: Neural network. Reprinted from [13]

3.2 Deep learning introduction

In our experiments we have used different varieties of deep neural network architectures. The central theme of deep learning architectures is a neural network. Neural network is loosely modeled on the human nervous system (shown in Fig 3.1). Classification problems are typically solved using such a network e.g the input is an image and the output is a label indicating whether the image contains a cat or a dog. We proceed to give a high-level introduction to the functioning of neural networks followed by a list of computation steps for a network.

A neural network model has two phases: training and testing. In Fig 3.1, on the left side, we

have an input vector followed by a hidden layer and on the right we have the output vector. Every interconnection in the graph has a weight w_{ij} associated with it (which is initialized randomly). Every node has a function associated with it (referred to in literature as activation function).

During training phase, initially, the signal flows from the left to right. At every node, the incoming signals are added, then the function value is computed and this value is propagated forward. At the output, the deviation of predicted output from the true output is calculated (using a loss function like cross entropy). The objective of the neural network is to optimize this loss function. This optimization is done using gradient descent method or its variants, for multiple iterations. The optimization algorithm is fed with entire data-set at every iteration (referred to in literature as an epoch). The weights are updated after every epoch.

In the testing phase, network takes in a previously unknown sample and generates a prediction. The final output of a neural network is a probability vector specifying the probability for every probable output label. A deep neural network is simply a network with multiple hidden layers. A DNN is able to learn arbitrarily complex functions (feature representations) due to introduction of multiple non-linearities.

Consider the illustrative neural network in Fig 3.1. The steps involved in the computation are as follows:

1. Input: 1-d vector 'x' with 3 entries x_i for i = 1,2,..3. (x_0 is the bias term).

2. Hidden layer: The network has one hidden layer with 3 nodes a_i for i = 1, 2,3. (a_0 is the bias term).

3. Output layer = $\hat{y}$; True output = y_{true}.

4. Weights: w_{ij}

5. Hidden node value computation: Assume that the activation function at hidden node is denoted by f_{act}. Consider the hidden node 1 in hidden layer. The input to this node is the

summation of all incoming signals:

Input = $\sum_{n=1}^{n} x_i * w_{ij}$;

Output = (a_i) = $f_{act}(input)$

6. Loss at output node: - $\sum_i y_{true} * log\hat{y}$ where the summation is over all possible output labels.

7. Minimize : loss

8. Output of network: p(y_i) = a_i for i = 1,2,...number of output labels

Varieties of deep neural network include the convolutional neural network (CNN) and recurrent neural network (RNN). Both the varieties of DNN have different characteristics. CNN's are useful in many computer vision tasks (spatial) while RNN's are useful in language processing tasks (temporal). RNN's are able to model dependencies present in sequential data in a better manner. We employ a variant of RNN known as long-short term memory network (LSTM) as well as CNN for our experiments. The input text data for all of the following experiments was taken from Project Gutenburg, an openly available text corpus [14]. We use Keras, an open source Python API for running and experimenting with various DNN architectures [15]. Training a deep neural network with right parameters takes multiple experiments. We use a GPU server for faster experimentation and for using bigger datasets to train the network.

4. LEARNING TEXT DECOMPRESSION

4.1 Why learn decompression for compressed text ?

Typically, characters are encoded using either fixed length or variable length encoding or any other encoding variants. The decoding of codewords depends on the availability of the codebook. We consider the scenario when the codebook is not available. During such cases, we want to do decoding using deep neural networks. Learning codebooks can be crucial when we have a file encoded with unknown encoding (e.g certain file-types like Adobe files have unknown encodings). When such a file encounters corrupted data, the learned codebook can then help in doing error correction. Research in this direction can help in building end-to-end decoding systems.

For our research, the input to the network is a binary sequence obtained after Huffman coding based compression of text. The network's output is the prediction of the corresponding decompressed character sequence. Specifically, the network is only provided with the input bit sequence (code-book is not available to the network). Essentially, the network has to learn the code-book that map bits to character as well as the individual code-word boundaries in the input sequence.

4.2 Learning to decompress texts without noise

Input: Input text compressed using Huffman code into a binary sequence of length 'n'.
Assumption: The first bit of binary sequence is also the first bit of a Huffman codeword. Every codeword encodes a single character.
Output: Sequence of characters whose Huffman codewords are wholly included in the binary sequence.

4.2.1 Notations

We introduce some standard notations for all of the following experiments.

1. n: binary input sequence length

2. m: character sequence output length

3. $|V|$: text vocabulary size

4.2.2 Output-concatenation model

We use a single neural network architecture for this task. For illustration purposes, consider a synthetic text consisting of only 'v' characters and an end of sequence special character: ('a','b','c','d','e','f','EOS') i.e $|V| = v + 1$. (Text generation is done using two models, random and Markovian. Markovian model is a better estimate for language data). If there are fewer than 'm' characters whose Huffman codewords are wholly included in the output sequence, we append 'EOS' characters to make the output sequence length = 'm'. Here we make sure that 'm' is sufficiently large so that the above approach is always feasible.

As with all Keras models, we use numeric label in output. For output-concatenation model, we one hot encode every character as follows:

1. 'a': [1, 0, 0, 0, 0, 0, 0]

2. 'b': [0, 1, 0, 0, 0, 0, 0]

3. 'c': [0, 0, 1, 0, 0, 0, 0]

4. 'd': [0, 0, 0, 1, 0, 0, 0]

5. 'e': [0, 0, 0, 0, 1, 0, 0]

6. 'f': [0, 0, 0, 0, 0, 1, 0]

7. 'end of sequence' (EOS): [0, 0, 0, 0, 0, 0, 1]

Our output sequence is obtained by concatenating 'm ' one-hot encoded vectors (length = $|V|$) to form one single output vector of length = $m * |V|$. Thus, this output vector is a binary vector. The network does prediction for every element value in this vector. The concatenated vector is

later mapped back to get the sequence (length = 'm') of predicted characters (using the one-hot mapping from above).

4.2.2.1 CNN-based model

We first employ a CNN based architecture for this task. The structure of the architecture is as follows (shown in Fig 4.1):

1. Input: Binary input sequence
2. 1D Convolutional layer: filter = 128 (refers to the number of feature maps generated by the layer), kernel_size = 5, stride length = 1, activation = 'relu'.
3. 1D max pooling: pooling size = 2
4. Flatten layer
5. Dense layer: 784 nodes, activation = 'relu'
6. Dense layer: $m * |V|$ nodes, activation = 'sigmoid'
7. Output: concatenated vector of length $m * |V|$
8. Loss function: Binary cross entropy

4.2.2.2 DNN-based model

We also built a purely DNN based architecture for this task with the following structure (shown in Fig 4.2):

1. Input: Binary input sequence
2. Dense layer: Nodes = 784, activation = 'relu'
3. Dense layer: Nodes = 100, activation = 'relu'
4. Dense layer: $m * |V|$ nodes, activation = 'sigmoid'

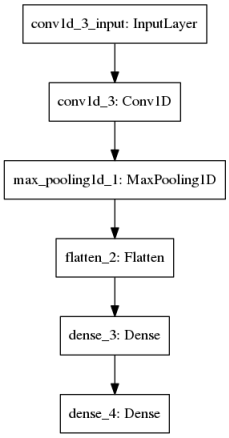

Figure 4.1: Output-concat CNN model for learning text decompression in noiseless texts

5. Output: concatenated vector of length $m * |V|$

6. Loss function: Binary cross entropy

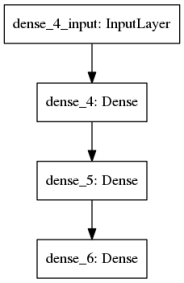

Figure 4.2: Output concat-DNN model for learning text decompression in noiseless texts

4.2.2.3 Results

Accuracy is defined as fraction of mapped output characters predicted correctly across all output samples. The datasets consist of a total of t = 1.5 M characters. We test both the models on the synthetic text data-sets with varying vocabulary sizes. The unnormalized frequency (# occurrences) of each character in the vocabulary is approximately equal to t / $|V|$. For this experiment, we have following parameters:

1. Input sequence length (n) = 100

2. Output character sequence length (m) = 100

3. Vocabulary size $|V|$ = 5, 10, 20, 40

The network learned the bit-to-character encoding perfectly for the given vocabulary sizes. When we trained the network using English language data-sets, the network output sequence consists of similar output labels. To support this claim, we observed the performance of the network over multiple epochs. The accuracy remained the same after successive epochs. We attribute this to two reasons. First, English language data-sets might not contain sufficient occurrences of each character for the network to learn properly. Second, such data-sets might need more complex models to effectively learn the decompression.

4.2.3 Branched output-layer model

We use a single DNN/CNN model for this task. Consider the input as a binary sequence of 'n' bits and the output from the model as a sequence of 'm' elements. We consider a synthetic text consisting of only 6 characters and a end of sequence special character: ('a','b','c','d','e','f','EOS') i.e $|V| = 7$.

4.2.3.1 Character to numeric dictionary

As this model directly gives an output character prediction in contrast to the earlier model (output-concatenation model) we use numeric labels in output. So we map every character to a numeric label as follows:

1. 'a': 0

2. 'b': 1

3. 'c': 2

4. 'd': 3

5. ‘e’: 4

6. ‘f’: 5

7. ‘end of sequence’ (EOS): 6

So for e.g ‘1120355406’, corresponds to ‘bbcadffeaEOS’.

4.2.3.2 DNN based model

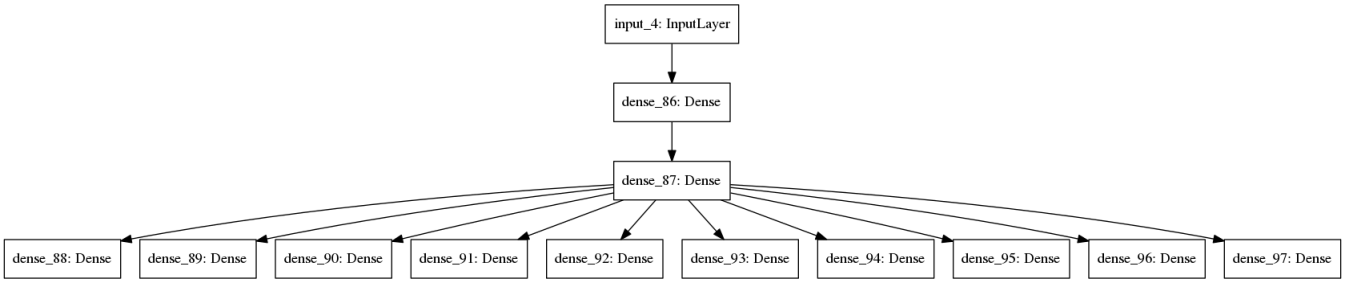

Figure 4.3: Branched DNN model for learning text decompression in noiseless texts

The structure of the single DNN model is as follows (consider the output to be a sequence of length ‘m’, shown in Fig 4.3):

1. Input layer: Binary input sequence

2. 1st hidden layer: Dense layer with 100 nodes. Activation = ‘relu’

3. 2nd hidden layer: Dense layer with 100 nodes. Activation = ‘relu’

4. After the 2nd hidden layer, the model has ‘m’ branches (each branch corresponds to one output element)

5. Output layer: Each of the ‘m’ branches is a Dense layer with ‘$|V|$’ nodes.
   (In the figure we have shown a model with m = 10 branches for illustrative purposes. We can set ‘m’ to any valid value of our choice.

6. Loss function: categorical cross entropy

7. Epochs: 20

8. Batch size: 16

9. Optimizer: RMSprop

The trained weights vector at the output layer is used to get the output prediction. The output prediction is a numeric sequence like '120355406'. This numeric sequence can be decoded to the character sequence ('bbcadffeaEOS') using the dictionary obtained in section 1.1. Accuracy is defined as fraction of output elements predicted correctly across all output samples. We include the results in the last section for easy comparison with other models.

4.2.3.3 CNN based model

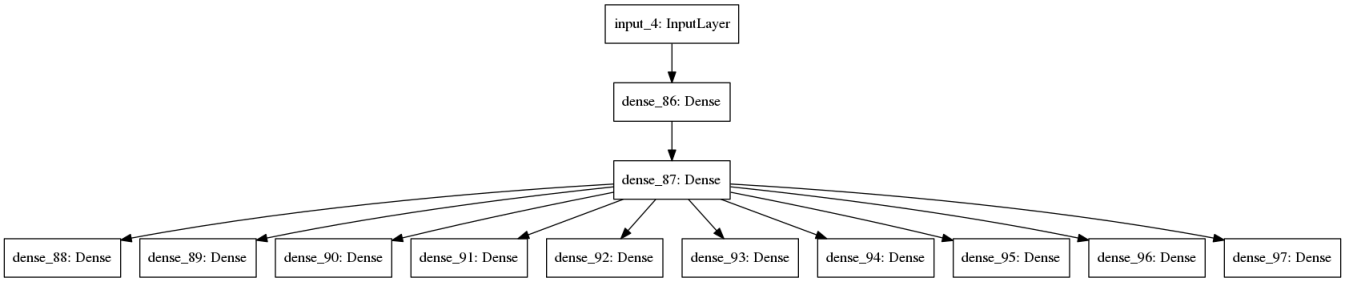

Figure 4.4: Branched CNN model for learning text decompression in noiseless texts

In these set of experiments we experiment with an architecture based on a single CNN. The structure of the model is as follows (consider the output to be a sequence of length 'm', shown in Fig 4.4):

1. Input: Binary input sequence

2. Convolutional 1D layer: filter = 32, kernel size = 3, stride length = 1, activation = 'relu'

3. Convolutional 1D layer: filter = 32, kernel size = 2, stride length = 1, activation = 'relu'

4. Flatten layer

5. Dense layer: Nodes = 100, activation = 'relu'

6. Dense layer: Nodes = 100, activation = 'relu'

7. Dense layer: Nodes = 100, activation = 'relu'

8. Dense layer: Nodes = 100, activation = 'relu'

9. At this point, the model has 'm' branches (each branch corresponds to one output element, just like the DNN model)

10. Each of the 'm' branches has a Dense layer on it with $|V|$ nodes.
    (In the figure we have shown a model with m = 10 branches for illustrative purposes. We can set 'm' to any valid value of our choice).

11. Loss function: Categorical cross entropy

12. Epochs: 20

13. Batch size: 16

14. Optimizer: RMSprop

4.2.3.4 Results

Accuracy is defined as fraction of output elements predicted correctly across all output samples. We test both the models on the synthetic text data-set with varying vocabulary sizes. The datasets consist of a total of t = 1.5 M characters. The unnormalized frequency (# occurrences) of each character in the vocabulary is approximately equal to t / $|V|$. We tabulate the results comparing the DNN model and CNN model (for vocabulary size = 40) in Table 4.1.

Model	Training accuracy	Testing accuracy
DNN	0.834	0.821
CNN	0.839	0.817

Table 4.1: Branched model comparison for learning compression in noiseless texts.

4.3 Learning to decompress text in presence of noise

Input: Input text compressed using Huffman code into a binary sequence of length 'n'. The binary sequence has bits erased randomly with probability 'p' (where $0 \leq p \leq 1$).

Assumption: The first bit of binary sequence is also the first bit of a Huffman codeword. Every codeword encodes a single character.

Output: Sequence of characters whose Huffman codewords are wholly included in the binary sequence.

4.3.1 CNN based model

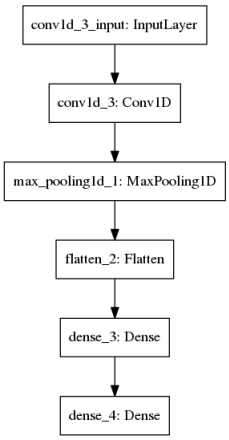

Figure 4.5: CNN model for learning text decompression in noisy texts

In these set of experiments we experiment with an architecture based on a single CNN. The structure of the model is as follows (consider the output to be a sequence of length 'm', shown in Fig 4.5):

vocabulary size	Testing erasure accuracy
5	0.9957
10	0.9912
20	0.9811
40	0.5378

Table 4.2: CNN model performance for learning compression in noisy texts.

1. Input: Binary input sequence
2. 1D Convolutional layer: filter = 128 (refers to the number of feature maps generated by the layer), kernel_size = 5, stride length = 1, activation = 'relu'.
3. 1D max pooling: pooling size = 2
4. Flatten layer
5. Dense layer: $m * |V|$ nodes, activation = 'sigmoid'
6. Output: concatenated vector of length $m * |V|$
7. Loss function: Binary cross entropy

4.3.2 Results

Accuracy is defined as fraction of output elements predicted correctly across all output samples. We test both the models on the synthetically generated text (Markov process generated) with varying vocabulary sizes. The datasets consist of a total of t = 1.5 M characters. The unnormalized frequency (# occurrences) of each character in the vocabulary is approximately equal to t / $|V|$. The model shows a limited performance on actual English language data-sets. For this experiment bit erasure probability 'p' = 0.2. We tabulate the results in Table 4.2.

4.4 Learning character encodings + boundaries

Input: Input text file encoded using Huffman coding into a binary sequence of length 'n'.

Assumption: The first bit of binary sequence is also the first bit of a Huffman codeword. Every codeword encodes a single character.

Output: Sequence of characters whose Huffman codewords are wholly included in the binary sequence + sequence indicating codeword boundaries (Binary sequence with 1's at codeword boundaries and 0's elsewhere.)

4.4.1 Notations

1. n: binary input sequence length
2. m: character sequence output length
3. $|V|$: output character vocabulary size
4. m*V + n: complete output length

We use a single DNN model for this task. For illustration, we consider a synthetic text consisting of only 6 characters and a end of sequence special character: ('a','b','c','d','e','f','EOS') i.e $|V| = 7$. If there are fewer than 'm' characters whose Huffman codewords are wholly included in the output sequence, we append 'EOS' characters to make the output sequence length = 'm'. Here we make sure that 'm' is sufficiently large so that the above approach is always feasible.

Again, we one-hot encode all characters as follows (similar to the output-concatenation model):

1. 'a': [1, 0, 0, 0, 0, 0, 0]
2. 'b': [0, 1, 0, 0, 0, 0, 0]
3. 'c': [0, 0, 1, 0, 0, 0, 0]
4. 'd': [0, 0, 0, 1, 0, 0, 0]

5. ‘e’: [0, 0, 0, 0, 1, 0, 0]

6. ‘f’: [0, 0, 0, 0, 0, 1, 0]

7. ‘end of sequence’ (EOS): [0, 0, 0, 0, 0, 0, 1]

Our output sequence is obtained by concatenating ‘m ’ one-hot encoded vectors (length = $|V|$) to form one single vector of length = $m * |V|$. This binary vector is then appended with the vector indicating codeword boundaries (Binary sequence with 1’s at codeword boundaries and 0’s elsewhere). The length of the complete output vector is thus, $m * V + n$. The network does prediction for every element value in this vector.

4.4.2 CNN based model

We employ a CNN based architecture for this task. The structure of the architecture is as follows (shown in Fig 4.6):

1. Input: Binary input sequence

2. 1D Convolutional layer: filter = 128 (refers to the number of feature maps generated by the layer), kernel_size = 5, stride length = 1, activation = ‘relu’.

3. 1D max pooling: pooling size = 2

4. Flatten layer

5. Dense layer: 784 nodes, activation = ‘relu’

6. Dense layer: $m * |V| + n$ nodes, activation = ‘sigmoid’

7. Output: concatenated vector of length $m * |V| + n$

8. Loss function: Binary cross entropy

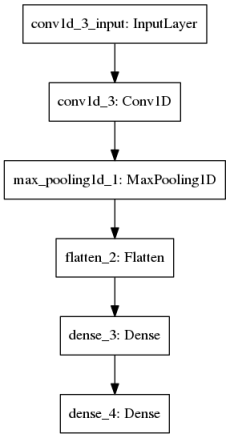

Figure 4.6: CNN model for codeword boundary recognition

4.4.3 DNN based model

We also built a purely DNN based architecture for this task with the following structure (shown in Fig 4.7):

1. Input: Binary input sequence
2. Dense layer: Nodes = 784, activation = 'relu'
3. Dense layer: Nodes = 100, activation = 'relu'
4. Dense layer: $m * |V| + n$ nodes, activation = 'sigmoid'
5. Output: concatenated vector of length $m * |V| + n$
6. Loss function: Binary cross entropy

4.4.4 Results

Accuracy is defined as fraction of mapped output elements predicted correctly across all output samples. Having built a model with m*V outputs (for bit to character mapping), we now test the codeword boundary prediction part accuracy (the 'n' part in $m * V + n$). We test both the models on the synthetically generated text (Markov process generated) with varying vocabulary sizes. The

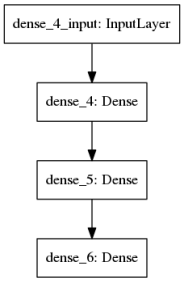

Figure 4.7: DNN model for codeword boundary recognition

datasets consist of a total of t = 1.5 M characters. The unnormalized frequency (# occurrences) of each character in the vocabulary is approximately equal to t / $|V|$. We test the models on the synthetic dataset with following parameters

1. Input sequence length (n) = 100
2. Vocabulary size $|V|$ = 7
3. Output sequence length ($m * V$ + n) = $100 * 7$ + 100 = 800

On this dataset, both DNN and CNN achieve near perfect results for predicting the codeword boundary sequence (thereby putting a good estimate on the codeword boundaries). The models do not perform well on datasets with higher vocabulary size, hinting that the network might be working for shorter length codewords.

5. LEARNING BIT ERASURE CORRECTION

Firstly, we present models for bit erasure correction in uncompressed ASCII encoded texts. Following that, we present models for learning erasure correction in Huffman compressed texts.

5.1 Bit erasure correction in uncompressed texts

5.1.1 Fixed bit erasure model

Input: Input bit sequence of length 'n' with the assumption that every multiple of n+1'th bit is erased (where n ϵ $\mathbb{N}$). 1/(n + 1) can be interpreted as the erasure probability.
Output: Network prediction for every bit erasure in the input sequence.

In this set of experiments we assume that every multiple of n'th bit is erased (where n is any natural number). In our experiment, we set 'n' = 100. In a way, (1/n) is the erasure probability. The text files used are taken from Project Gutenburg. We employ a deep neural network to predict the values of erased bits alone. We use a variant of recurrent neural network (mentioned earlier) for this experiment. This variant is known as long-short term memory network (LSTM). LSTMs have been shown to work well for language processing tasks [16]. We use certain other machine learning algorithms: SVM, random forest as well.

Among all models, LSTM gives a better performance. We measure the test accuracy by calculating the fraction of bit erasure values that the network decodes correctly. LSTM network gives an accuracy of 0.8465. This model assumes that every multiple of n'th bit is erased. Actual noisy channel assumes random noise. So this experiment serves only as a starting point for our further experiments.

5.1.2 Random bit erasure model

We work with uncompressed, ASCII encoded text in this model.
Input: ASCII encoded binary sequence where bits are erased randomly with probability 'p' (where $0 \leq p \leq 1$).

Output: Network prediction for every bit erasure in the input sequence.

5.1.2.1 RNN based model

In this set of experiments we assume random erasure of bits with probability 'p' (where $0 \leq p \leq 1$). LSTMs (variant of RNN) have been shown to work well for language processing tasks [16]. Particularly, we model this problem as a sequence to sequence learning problem as outlined by Sutskever et al in [7]. In our case, the input sequence is a bit sequence with erasures and the output sequence by the network is the sequence with erasure values predicted. These type of problems require an encoder-decoder network model consisting of a stack of LSTM layers. Every LSTM has an internal state associated with it. LSTM layer has three outputs: LSTM final output, internal cell output and hidden state output. We describe the model below for completeness.

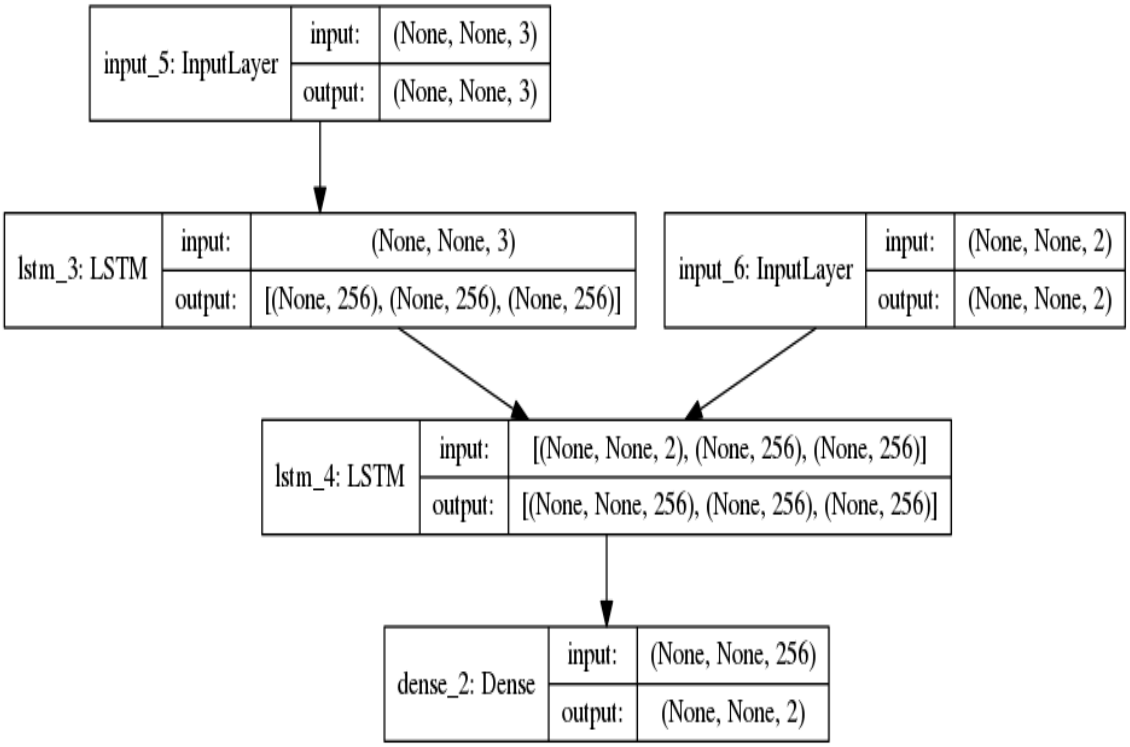

Figure 5.1: Encoder-decoder model for bit erasure correction in uncompressed text

We use an encoder-decoder model that has shown impressive results for sequence to sequence learning. The input-output connections and layers as shown in Fig 5.1 are as follows:

1. input_5 (in Fig 5.1) : Input bit sequence with erasures. We consider the input bits to have 3 distinct values: 0, 1 or erasure.

2. input_6 (in Fig 5.1) : Offset version (1 time-step in forward direction) of true output bit sequence.

   (Available only during training phase to initialize weights for the model. During testing phase we start the prediction with only a start of sequence token and the hidden state output from lstm_3 (encoder LSTM). In our case, the start token is a bit. This start bit is set arbitrarily only for the purpose of getting the prediction process started during testing phase. The decoder model recursively appends one bit at a time to the output sequence).

3. lstm_3 : Encoder LSTM

4. lstm_4 : Decoder LSTM

5. dense_2 ; Dense layer to get final output from network. Output bits have 2 label values for the present model. (0 or 1).

Encoder model

Keras LSTM models require the inputs and outputs arrays to be of the shape [number of samples, time instants in one input sequence = sequence length, features for one time instant = 3]. We encode every bit as a one-hot encoded variable (giving us feature array of length 3 for each time instant):

1. one hot encoding for 0 : [1 , 0 , 0]

2. one hot encoding for 1 : [0 , 1, 0]

3. one hot encoding for erasure : [0 , 0, 1]

In Fig 5.1, 'input_5' is the array of input sequences with erasures with shape [number of samples in dataset, sequence length, 3]. 'input_5' is the input to the encoder LSTM ('lstm_3'). The encoder LSTM computes its internal hidden state, internal cell state along with output. We need only the hidden state and cell state to condition the decoder LSTM network in the next stage.

Decoder model

Next stage is the decoder model. 'input_6' is the shifted version (by 1 bit in the forward direction) of the true output sequence. Since this is the shifted true output sequence, it has 0's and 1's only. 'input_6' has shape [number of samples in dataset, sequence length, 2].

The input to decoder LSTM, ('lstm_4') is 'input_6' (of shape [None,None,2]) and the hidden state and cell state output (of shape [None,None,256]) of encoder LSTM ('lstm_3'). The hidden state of encoder LSTM is used to initialize internal state of decoder LSTM. At a time instant 't', the decoder LSTM takes in as its input the true output up to time 't' and conditioned on the hidden state output from encoder LSTM, it gives a prediction for output at time 't+1'.

Dense layer

The dense layer takes in as input, the output from decoder LSTM and generates an output prediction for every time-step. Output has 2 labels: 0 and 1 (erased bit's prediction)

The random erasure model is a realistic representation of an actual communication channel. We used text files from Project Gutenburg (online text corpus) for our experiments. The training set consisted of 1.5 M samples (32.0 MB of text data). The testing set consisted of 300k samples (5 MB of text data). We trained the network for 5 epochs (5 runs over entire training set). We did a detailed experimentation for the parameter tuning by recording model performance for different parameters.

Accuracy is defined as the fraction of erased bits values that the model is able to predict correctly. We used a text dataset from Project Gutenburg. The bit erasure probability 'p' = 0.2. The network has an accuracy of 0.8384.

5.1.2.2 CNN based model

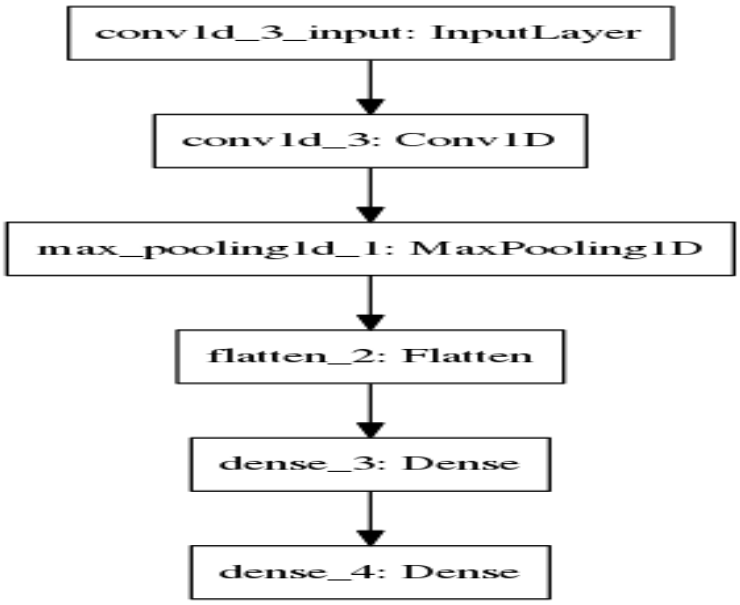

Figure 5.2: CNN model for bit erasure correction in uncompressed text

In this subsection, we outline the usage of CNN for the random erasure model. This model shows a performance similar to that of the RNN based model. While RNN is said to be better suited for language processing tasks, it is computationally expensive. CNN model has lesser computation requirements.

The CNN based architecture is as follows (shown in Fig 5.2):

1. Input layer: Binary sequence with random erasures
2. Convolutional 1D layer: 128 feature maps, kernel_size = 5
3. MaxPooling 1D layer: pool_size = 2
4. Flatten layer
5. Dense layer

6. Dense layer

Accuracy is defined as the fraction of erased bits values that the model is able to predict correctly. We used a text dataset from Project Gutenburg. The bit erasure probability 'p' = 0.2. This network has an accuracy of 0.8411.

5.2 Bit erasure correction in compressed texts

Input: Input text file encoded using Huffman coding into a binary sequence of length 'n'. In the binary sequence, bits are erased randomly with probability 'p' (where $0 \leq p \leq 1$).

Assumption: The first bit of binary sequence is also the first bit of a Huffman codeword. Every codeword encodes a single character.

Output: Binary sequence with prediction for erasures.

5.2.1 Notations

1. n: binary input sequence length
2. m: character sequence output length
3. $|V|$: output character vocabulary size
4. p: Bit erasure probability

5.2.2 3-sequential CNN architecture

We use a 3-sequential CNN architecture for this task. That is, the architecture consists of 3 CNN's stacked sequentially. Output from a CNN is the input for the subsequent CNN. We train / test the entire architecture by just feeding input to the first CNN and then subsequently obtaining outputs to be used as inputs for the next CNN.

For illustration purposes, gain let's consider a synthetic text consisting of 6 characters and a end of sequence special character: ('a','b','c','d','e','f','EOS') i.e $|V| = 7$. If there are fewer than 'm' characters whose Huffman codewords are wholly included in the output sequence, we

append ‘EOS’ characters to make the output sequence length = ‘m’. Here we make sure that ‘m’ is sufficiently large so that the above approach is always feasible.

We one hot encode every character as follows (similar to output-concatenation model):

1. ‘a’: [1, 0, 0, 0, 0, 0, 0]
2. ‘b’: [0, 1, 0, 0, 0, 0, 0]
3. ‘c’: [0, 0, 1, 0, 0, 0, 0]
4. ‘d’: [0, 0, 0, 1, 0, 0, 0]
5. ‘e’: [0, 0, 0, 0, 1, 0, 0]
6. ‘f’: [0, 0, 0, 0, 0, 1, 0]
7. ‘end of sequence’ (EOS): [0, 0, 0, 0, 0, 0, 1]

Our output sequence for characters is obtained by concatenating ‘m ’ one-hot encoded vectors (length = $|V|$) to form one single output vector of length = $m * |V|$. Thus, this vector is a binary vector. The network does prediction for every element value in this vector. The concatenated vector is later mapped back to get the sequence (length = ‘m’) of predicted characters (using the one-hot mapping from above). Our 3-sequential CNN architecture is outlined below.

5.2.2.1 1st CNN

The 1st CNN’s use is to learn a mapping from binary sequence with erasures to characters. The structure of the 1st CNN is as follows (shown in Fig 5.3):

1. Input: Input bit sequence with erasures. Erasure is represented by ‘-1’
2. 1D Convolutional layer: filter = 256 (refers to the number of feature maps generated by the layer), kernel_size = 5, stride length = 1, activation = ‘relu’.
3. 1D max pooling: pooling size = 2

4. Flatten layer

5. Dense layer: 784 nodes, activation = ‘relu’

6. Dense layer: $m * |V|$ nodes, activation = ‘sigmoid’

7. Output: concatenated vector of length $m * |V|$

8. Loss function: Binary cross entropy

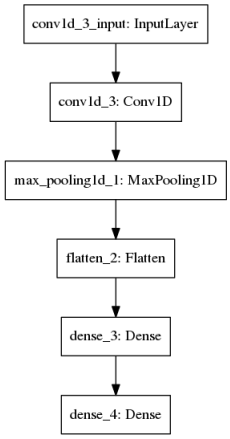

Figure 5.3: 1st CNN in 3-sequential model for bit erasure correction in compressed text

5.2.2.2 2nd CNN

The 2nd CNN takes the output from 1st CNN as its input. The output of this network is a character sequence. The 1st CNN may have errors at the output. So the network may not be able to fully learn the replacement and insertion / deletion errors for the character sequence. The (2nd) CNN’s purpose is to learn these errors. Structure of this CNN is as follows (shown in Fig 5.4):

1. Input: Sequence outputted by 1st CNN

2. 1D Convolutional layer: filter = 256 (refers to the number of feature maps generated by the layer), kernel_size = 5, stride length = 1, activation = ‘relu’.

3. 1D max pooling: pooling size = 2

4. Flatten layer

5. Dense layer: 784 nodes, activation = ‘relu’

6. Dense layer: $m * |V|$ nodes, activation = ‘sigmoid’

7. Output: concatenated vector of length $m * |V|$

8. Loss function: Binary cross entropy

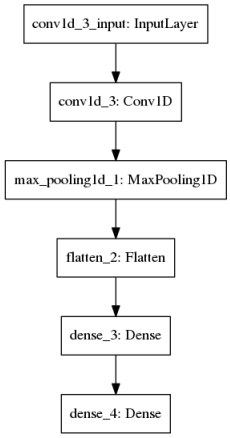

Figure 5.4: 2nd CNN in 3-sequential model for bit erasure correction in compressed text

5.2.2.3 3rd CNN

The 3rd CNN takes the output from 2nd CNN as its input. The output of this network is a binary sequence with the same length as binary input sequence. The use of this CNN is to learn the erasures in the original sequence and generate predictions for those elements at the bit level. This CNN reconstructs the input sequence with predictions for erased bits. The structure of the CNN is as follows (shown in Fig 5.5):

1. Input: Sequence outputted by 2nd CNN.

2. 1D Convolutional layer: filter = 256 (refers to the number of feature maps generated by the layer), kernel_size = 5, stride length = 1, activation = 'relu'.

3. 1D max pooling: pooling size = 2

4. Flatten layer

5. Dense layer: 784 nodes, activation = 'relu'

6. Dense layer: n nodes, activation = 'sigmoid'

7. Output: Binary sequence of length = 'n'

8. Loss function: Binary cross entropy

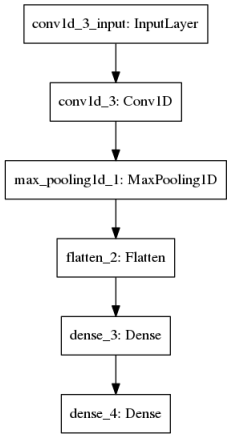

Figure 5.5: 3rd CNN in 3-sequential model for bit erasure correction in compressed text

5.2.3 Results

Accuracy is defined as fraction of erased bits predicted correctly across all output samples. For this experiment, we set bit erasure probability = 0.2. We tested the architecture on synthetic data-set (generated by Markov model) with varying vocabulary sizes. The datasets consist of a total of t = 1.5 M characters. The probabilities in the Markov model are equal to the normalized frequencies

Vocabulary size	erasure accuracy
5	0.935
10	0.921
20	0.812
40	0.775

Table 5.1: Bit erasure prediction in compressed texts.

of the characters in the vocabulary (# occurrences of character / t). We tabulate the results in Table 5.1. For this experiment, we have following parameters:

1. Input sequence length (n) = 100
2. Output character sequence length (m) = 100
3. Vocabulary size $|V|$ = 5, 10, 20 and 40
4. Bit erasure probability = 0.2

6. CONCLUSION AND DISCUSSION

6.1 Decompression for compressed texts

In Chapter 4, we outlined models for learning decompression for texts compressed by Huffman code. We described the need for building such models in 4.1. Such models can help with error correction when the codebook is unavailable to the system. A network learning to decompress text has to learn the bit to character mapping as well as the individual codeword boundaries. Since the network that we built had only the bits as input and not the codebook, the model show promise for learning decompression for other compression algorithms as well. Since majority of compression algorithms are structured, the real test of the network lies in its ability to learn the structure of the encoding.

We have built a CNN based as well as DNN based network architecture for this task. Firstly, we trained our network on two types of data. One was text data generated from random model. Second was text data generated from a Markovian model (this model is a better realization of language data). We varied the vocabulary size of our input text, starting from values = 5,10 up to 40. The network was able to learn the encodings for the data generated by above models for all these vocabulary sizes. Model building included data processing to bring the data in the right format, followed by structural optimization including hyper-parameter tuning to obtain the best performance.

For compression algorithms like Huffman coding, the variable length codewords pose a challenge to the network. We deal with this challenge as follows: We set the # feature maps to a number greater the input binary sequence length. The CNN has a 'kernel size' parameter which specifies the window size when computing the feature maps. We set the value of this parameter to a large enough value, so that the network is able to learn all possible length encodings. We set the value of the 'stride length' parameter = 1, so that the network does not miss out any codeword boundary. When the window slides through the input binary sequence, it looks at all possible codeword com-

binations. After successive epochs, the network learns the codebook. In 4.4, we also investigate codeword boundary prediction separately. The network is able to learn the codeword boundaries

6.2 Bit erasure correction

In 5.1, we outlined models for bit erasure prediction in uncompressed texts. Typically, error correction models involve adding external redundancy in the data that is transmitted across a noisy channel. At the receiver end, the error-correcting code recovers the original data. The performance of error-correcting codes have are subject to certain information theoretic bounds / limits. In our research, we are building a system that works on the decoding part. Taking in data with erasures, we do erasure-correction. An important point to be noted is that the input binary sequence has no artificial redundancy. The prediction systems worked entirely on the natural redundancy present in the data.

The first model assumed erasures at fixed bit positions (code in Appendix A). This model only serves as a starting point for our experiments. Amongst all machine learning models, a deep neural network based model gave the best performance. Next, we outlined the random erasure model where every bit had a probability = 'p' of being an erasure (code in Appendix A). We employed both RNN as well as CNN based architecture for this task with similar results. Model building included data processing to get the data in the right format, followed by structural optimization including hyper-parameter tuning to obtain the best performance. Deep learning based models (RNN / CNN) outperform classical learning models like SVM, logistic classifier. This points towards the fact that deep learning based systems have more potential to learn arbitrarily complex function mappings as compared to classical learning models at the bit level.

For RNN-based architecture, we make use of the natural redundancy in the data as follows: We use a bi-directional LSTM (RNN variant). The network is able to capture the co-occurence relationship between consecutive bits / erasures. Whereas for a CNN, we set # hidden nodes in last Dense layer = # bits in output binary sequence. This leads to a correspondence between the hidden node value and the output bits. The intermediate stack of hidden layers in the network

have interconnections between nodes. This way, the deep neural networks learn to make use of the natural redundancy to do erasure correction.

Next step was to build a model for bit erasure correction in compressed texts. In 5.2, we explore models for this idea. Conceptually the network architecture can be thought of as follows: The input to the system is a binary sequence with erasures. The 1st CNN learns the bit-to-character mapping in presence of erasures. The output sequence of this CNN is inputted to the 2nd CNN. The 2nd CNN learns to map this sequence to the correct sequence (bit-to character mapping without erasures). The output of this CNN is inputted to the 3rd CNN. The 3rd CNN learns the mapping from characters back to a binary sequence. The final stage output is expected to give the prediction for input erasures. Model building included data pre-processing to bring the data in the right format, followed by structural optimization including hyper-parameter tuning to obtain the best performance.

The 3CNN architecture can be thought to be analogous to a encoder-decoder system. The 1st CNN learns the bit-to-char encoder. The 2nd CNN learns a character-to-character mapping (learning a noisy channel). The 3rd and final CNN, decodes the character sequence back to the binary sequence. We observe a good performance of this architecture when the vocabulary size is smaller. Smaller vocabulary size leads to a shorter length codewords.

The novelty of our research is two-fold. Firstly, we build a deep neural network based model for working directly with bit level language data. Research in the natural language processing community so far has focused only on word level / character level language data processing. We prefer to use deep neural net architectures to leverage the natural redundancy present in language data. The models can be investigated further to build efficient algorithms for handling language data in presence of erasures.

REFERENCES

[1] I. Tamo and A. Barg, "A family of optimal locally recoverable codes," *IEEE Transactions on Information Theory*, vol. 60, pp. 4661–4676, Aug. 2014.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] E. Nachmani, E. Marciano, L. Lugosch, W. J. Gross, D. Burshtein, and Y. Be'ery, "Deep learning methods for improved decoding of linear codes," *CoRR*, vol. abs/1706.07043, 2017.

[4] T. Gruber, S. Cammerer, J. Hoydis, and S. ten Brink, "On deep learning-based channel decoding," *CoRR*, vol. abs/1701.07738, 2017.

[5] J. Brownlee, *Deep Learning with Python*. http://www.machinelearningmastery.org.

[6] Y. Shen, H. Yun, Z. C. Lipton, Y. Kronrod, and A. Anandkumar, "Deep active learning for named entity recognition," in *Proceedings of the 2nd Workshop on Representation Learning for NLP, Rep4NLP@ACL 2017, Vancouver, Canada, August 3, 2017* (P. Blunsom, A. Bordes, K. Cho, S. B. Cohen, C. Dyer, E. Grefenstette, K. M. Hermann, L. Rimell, J. Weston, and S. Yih, eds.), pp. 252–256, Association for Computational Linguistics, 2017.

[7] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3104–3112, 2014.

[8] E. Nachmani, Y. Be'ery, and D. Burshtein, "Learning to decode linear codes using deep learning," *CoRR*, vol. abs/1607.04793, 2016.

[9] N. Farsad, M. Rao, and A. Goldsmith, "Deep learning for joint source-channel coding of text," *CoRR*, vol. abs/1802.06832, 2018.

[10] A. Jiang, P. Upadhyaya, E. F. Haratsch, and J. Bruck, "Correcting errors by natural redundancy," in *2017 Information Theory and Applications Workshop, ITA 2017, San Diego, CA, USA, February 12-17, 2017*, pp. 1–8, IEEE, 2017.

[11] P. Upadhyaya and A. A. Jiang, "On LDPC decoding with natural redundancy," in *55th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2017, Monticello, IL, USA, October 3-6, 2017*, pp. 680–687, IEEE, 2017.

[12] Y. Wang, M. Qin, K. R. Narayanan, A. Jiang, and Z. Bandic, "Joint source-channel decoding of polar codes for language-based source," *CoRR*, vol. abs/1601.06184, 2016.

[13] S. Raschka, *Python Machine Learning*. Birmingham, UK: Packt Publishing, 2015.

[14] *Project Gutenburg*. http://www.gutenburg.org.

[15] F. Chollet, "keras." `https://github.com/fchollet/keras`, 2015.

[16] K. Yao, B. Peng, Y. Zhang, D. Yu, G. Zweig, and Y. Shi, "Spoken language understanding using long short-term memory neural networks," in *2014 IEEE Spoken Language Technology Workshop, SLT 2014, South Lake Tahoe, NV, USA, December 7-10, 2014*, pp. 189–194, IEEE, 2014.

APPENDIX A

BIT ERASURE CORRECTION MODELS

This appendix contains the code for erasure correction models [5]:

A.1 Fixed erasure model implementation

```
#bit level erasure correction using LSTM network architecture
import sys
import numpy
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM
from keras import metrics
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils

#Snippet to convert ASCII encoded file content to binary bit
   stream
binary = ''
with open('concat.txt') as f:
  while True:
    c = f.read(1)
    if not c:
      print("End of file")
      break
    binary = binary + bin(ord(c))[2:]
```

```
n_chars = len(binary)
print("Total␣Characters:␣", n_chars)

#Prepare the dataset of input to output pairs encoded as integers
seq_length = 99
dataX = []
dataY = []
for i in range(0, int(n_chars/seq_length), 1):
    seq_in = binary[i*seq_length:(i+1)*seq_length]
    seq_in = [int(x) for x in seq_in]
    seq_out = binary[(i+1)*seq_length]
    dataX.append(seq_in)
    dataY.append(int(seq_out))
n_patterns = len(dataX)
print("Total␣Patterns:␣", n_patterns)

# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
y = np_utils.to_categorical(dataY, num_classes=None)

# define the LSTM model
model = Sequential()
model.add(LSTM(256, dropout = 0.2, recurrent_dropout = 0.2,
   input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(LSTM(256, dropout= 0.4, recurrent_dropout = 0.1))
model.add(Dense(y.shape[1], activation='softmax'))
```

```
model.compile(loss='binary_crossentropy', optimizer='RMSprop',
    metrics=['acc'])
model.summary()
keras.utils.plot_model(model, to_file='model.png', show_shapes=
    True, show_layer_names=True, rankdir='TB')
callbacks_list = [checkpoint]


# fit the model
history = model.fit(X, y, validation_split = 0.2, epochs=50,
    batch_size=256, callbacks = callbacks_list)
```

A.2 Random erasure model implementation

```
#random erasure prediction in bit sequences
from __future__ import print_function
import sys
import numpy
from numpy import array, argmax, array_equal
import keras
import random
from keras.preprocessing.text import one_hot
from keras.models import Sequential, Model, Input
from keras.layers import Dense, Dropout, LSTM, Bidirectional
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
#from attention_decoder import AttentionDecoder
```

```
filepath_train = 'janeausten.txt'
filepath_test = 'wonderland.txt'
sequence_length = 4095
erasure_probability = 0.2
nb_passes_train = 1
nb_passes_test = 1

def text_to_binary(filepath):
    print("Converting␣text␣file␣to␣binary␣bit␣stream")
    binary = ''
    with open(filepath) as f:
        while True:
            c = f.read(1)
            if not c:
                print("\nEnd␣of␣file␣reading␣operation")
                break
            binary = binary + bin(ord(c))[2:]
    return binary

def data_preprocessing_decoder(binary):
    print("\nPre␣processing␣decoder␣data␣for␣model....")
    decoder_input, decoder_output = list(), list()
    temp = []
    for j in range(0, steps_per_dataset, 1):
        seq_out =
        [int(x) for x in binary[j*sequence_length:(j+1)*
           sequence_length]]
```

```
            temp.append(seq_out)
            decoder_output.append(np_utils.to_categorical(seq_out,
                num_classes = 2))
            decoder_input.append
            (np_utils.to_categorical(list([0]+seq_out[:-1]),
                num_classes = 2))
        return array(decoder_input), array(decoder_output),temp

def data_preprocessing_encoder(decoder_output_raw):
    print("\nPre processing encoder data for model....")
    input_seq = list()
    for i in range(steps_per_dataset):
        seq = decoder_output_raw[i]
        for j in range(0, sequence_length, 1):
            if random.random() < erasure_probability:
                seq[j] = -1
        input_seq.append(np_utils.to_categorical(seq, num_classes
            = 2))
    return array(input_seq)

# returns train, inference_encoder and inference_decoder models
def define_models(n_input, n_output, n_units):
    # define training encoder
    encoder_inputs = Input(shape=(None, n_input))
    encoder = LSTM(n_units, return_state=True)
    encoder_outputs, state_h, state_c = encoder(encoder_inputs)
    encoder_states = [state_h, state_c]
```

```
# define training decoder
decoder_inputs = Input(shape=(None, n_output))
decoder_lstm = LSTM(n_units, return_sequences=True,
   return_state=True)
decoder_outputs, _, _ =
decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(n_output, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
model = Model([encoder_inputs, decoder_inputs],
   decoder_outputs)
# define inference encoder
encoder_model = Model(encoder_inputs, encoder_states)
# define inference decoder
decoder_state_input_h = Input(shape=(n_units,))
decoder_state_input_c = Input(shape=(n_units,))
decoder_states_inputs = [decoder_state_input_h,
   decoder_state_input_c]
decoder_outputs, state_h, state_c =
decoder_lstm(decoder_inputs, initial_state=
   decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model =
Model([decoder_inputs]+decoder_states_inputs,
[decoder_outputs]+decoder_states)
#return all models
return model, encoder_model, decoder_model
```

```
# generate target given source sequence
def predict_sequence(infenc, infdec, source, n_steps, cardinality
    ):
        # encode
        state = infenc.predict(source)
        # start of sequence input
        target_seq = array([0.0 for _ in range(cardinality)]).
            reshape(1, 1, cardinality)
        # collect predictions
        output = list()
        for t in range(n_steps):
                # predict next char
                yhat, h, c = infdec.predict([target_seq] + state)
                # store prediction
                output.append(yhat[0,0,:])
                # update state
                state = [h, c]
                # update target sequence
                target_seq = yhat
        return array(output)

# decode a one hot encoded string
def one_hot_decode(encoded_seq):
        return [argmax(vector) for vector in encoded_seq]
original_sequence = text_to_binary(filepath_train)
print("\nLength␣of␣input␣sequence:␣", len(original_sequence))
```

```
steps_per_dataset = int(len(original_sequence)/sequence_length)
X2_tr, y_tr, decoder_output_raw = data_preprocessing_decoder(
    original_sequence)
X1_tr = data_preprocessing_encoder(decoder_output_raw)
print("\nTotal␣number␣of␣training␣samples:␣",X1_tr.shape[0])

n_features = 2
n_steps_in = sequence_length
n_steps_out = sequence_length

# define model
model, infenc, infdec = define_models(n_features, n_features,
    256)
model.compile(optimizer='RMSprop', loss='categorical_crossentropy
    ',
metrics=['acc'])

callbacks_list = [checkpoint]
model.fit([X1_tr, X2_tr],y_tr,validation_split = 0.2, epochs =
    20,verbose = 1,
batch_size = 16, callbacks = callbacks_list)

erasure = 0
prediction = 0

for i in range(X1_te.shape[0]):
    for j in range(X1_te.shape[1]):
```

```
        if X1_te[i,j,2] == 1.0:
            erasure = erasure + 1
            target = predict_sequence(infenc, infdec, X1_te[i].
               reshape((1,X1_te.shape[1],X1_te.shape[2])),
               sequence_length, n_features_out)
            #a = argmax(y_te[i,j])
            #b = one_hot_decode(target)
            if argmax(y_te[i,j]) == argmax(target[j]):
                prediction = prediction + 1

print("Test accuracy: ",prediction/erasure)
```

APPENDIX B

DATA PREPARATION FOR TEXT DECOMPRESSION

This appendix contains code for learning bit to character encodings. [5]

B.1 Raw text data pre-processing

```
from __future__ import print_function
import os
import sys
import random
import huffman
import collections
import keras
import numpy
from numpy import array
from numpy import argmax
from numpy import array_equal
from numpy.random import seed
from itertools import count
from collections import defaultdict
from bitarray import bitarray
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential, Model, Input
from keras.layers import Input, Dense, Flatten, Dropout
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import one_hot
```

```
from keras.callbacks import ModelCheckpoint, TensorBoard

filepath_train = 'wonderland.txt'
sequence_length = 50
erasure_probability = 0.2
raw_text = open(filepath_train).read()
#raw_text = 'a'*23345 + 'b'*31289 + 'c'*54665 + 'd'*12454 + 'e'
    '*55399 + 'f'*36738
#raw_text =''.join(random.sample(raw_text,len(raw_text)))
chars = sorted(list(set(raw_text)))
vocabulary_size = len(chars)+1
print("Size of training vocabulary: ", vocabulary_size)
char_to_int = dict((c,i) for i,c in enumerate(chars))
int_to_char = dict((i,c) for i,c in enumerate(chars))

#binary = ''
#with open('janeausten.txt') as f:
#  while True:
#    c = f.read(1)
#    if not c:
#      print("End of file")
#      break
#    binary = binary + bin(ord(c))[2:]

def text_encoding():
    dictionary = huffman.codebook(collections.Counter(raw_text).
        items())
```

```
    for item in dictionary:
        dictionary[item] = bitarray(dictionary[item])
    binary = bitarray()
    binary.encode(dictionary, raw_text)
    return binary,dictionary


binary,huffman_dict = text_encoding()
print('Length of raw text: ', len(raw_text))
print('Length of encoded text', len(binary))
```

B.2 Output-concatenation model data preparation

```
dataX = []
dataY = []
encoding_ratio = int(len(binary)/len(raw_text))
#output_length = int(sequence_length/encoding_ratio)
print("Average integer codeword length: ",encoding_ratio)
#print("Length of each output seqeunce: ", output_length)
output_length = 15


for i in range(0, int(len(binary)/sequence_length), 1):
    seq_in_array = binary[i*sequence_length:(i+1)*sequence_length
        ]
    seq_in = [int(x) for x in seq_in_array]
    seq_out = bitarray(seq_in_array).decode(huffman_dict)
    seq_out_encoded = [char_to_int[x] for x in seq_out]
    seq_out_encoded.extend([len(chars) for _ in range(
        output_length - len(seq_out_encoded))])
```

```
    dataY.append(seq_out_encoded)
    dataX.append(seq_in)

X= numpy.reshape(dataX,(len(dataX),sequence_length)) #input data
samples = X.shape[0]
print("Input shape: ", X.shape)
output = numpy.reshape(dataY,(len(dataY),output_length))  #output
    data
yconcat = np_utils.to_categorical(output,num_classes = None)
y = numpy.reshape(yconcat,(samples,(output_length)*(
   vocabulary_size)))
print("Output shape: ", y.shape)
```

B.2.1 Output-concatenation DNN model

```
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
import keras.backend as K


#train test split
X_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.3)
train_samples = X_tr.shape[0]
test_samples = X_te.shape[0]
#DNN model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim =
   sequence_length))
```

```
#model.add(Dropout(0.1))
#model.add(Dense(100, activation='relu'))
#model.add(Dense(100, activation='relu'))
#model.add(Dense(100, activation='relu'))
#model.add(Dense(100, activation='relu'))

#model.add(Dropout(0.1))
model.add(Dense(y.shape[1], activation='sigmoid'))

#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
#callbacks_list = [checkpoint]

model.fit(X_tr, y_tr, epochs= 5, batch_size=16, verbose=1)

#Calculate training and testing predictions
preds_tr = model.predict(X_tr)
preds_tr[preds_tr>=0.5] = 1
preds_tr[preds_tr<0.5] = 0
ypred_tr = numpy.reshape(preds_tr,(train_samples,output_length,
    vocabulary_size))
ypred_tr = numpy.argmax(ypred_tr, axis = 2)

preds_te = model.predict(X_te)
preds_te[preds_te>=0.5] = 1
```

```
preds_te[preds_te<0.5] = 0
ypred_te = numpy.reshape(preds_te,(test_samples,output_length,
    vocabulary_size))
ypred_te = numpy.argmax(ypred_te, axis = 2)

y_tr = numpy.reshape(y_tr,(train_samples,output_length,
    vocabulary_size))
y_tr = numpy.argmax(y_tr, axis = 2)

y_te = numpy.reshape(y_te,(test_samples,output_length,
    vocabulary_size))
y_te = numpy.argmax(y_te, axis = 2)

# Calculate training and testing accuracies
t = 0
correct = 0
for i in range(train_samples):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_tr[i,j] == ypred_tr[i,j]:
                correct = correct + 1
print("Training␣accuracy:␣", float(correct)/float(t))

t = 0
correct = 0
for i in range(test_samples):
```

```
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_te[i,j] == ypred_te[i,j]:
                correct = correct + 1
print("Testing accuracy: ", float(correct)/float(t))
```

B.2.2 Output-concatenation CNN model

```
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
import keras.backend as K
from keras.layers.convolutional import Conv1D
from keras.layers.pooling import MaxPooling1D

#train test split
X_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.2)
xtr = numpy.reshape(X_tr,(X_tr.shape[0],X_tr.shape[1],1))
xte = numpy.reshape(X_te,(X_te.shape[0],X_te.shape[1],1))

train_samples = X_tr.shape[0]
test_samples = X_te.shape[0]
#DNN model
model = Sequential()
model.add(Conv1D(filters = 32, kernel_size = 5, strides = 1,
   activation='relu', input_shape=(sequence_length,1)))
model.add(MaxPooling1D(pool_size = 4))
model.add(Conv1D(filters = 16, kernel_size = 3, strides = 1,
```

```
    activation='relu'))
model.add(MaxPooling1D(pool_size = 2))
model.add(Flatten())
model.add(Dense(784, activation='relu', input_dim =
    sequence_length))
#model.add(Dropout(0.1))
#model.add(Dense(100, activation='relu'))
#model.add(Dropout(0.1))
model.add(Dense(y.shape[1], activation='sigmoid'))

#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
#callbacks_list = [checkpoint]

model.fit(xtr, y_tr, epochs=4, batch_size=32, verbose=1)

#Calculate training and testing predictions
preds_tr = model.predict(xtr)
preds_tr[preds_tr>=0.5] = 1
preds_tr[preds_tr<0.5] = 0
ypred_tr = numpy.reshape(preds_tr,(train_samples,output_length,
    vocabulary_size))
ypred_tr = numpy.argmax(ypred_tr, axis = 2)

preds_te = model.predict(xte)
```

```
preds_te[preds_te>=0.5] = 1
preds_te[preds_te<0.5] = 0
ypred_te = numpy.reshape(preds_te,(test_samples,output_length,
    vocabulary_size))
ypred_te = numpy.argmax(ypred_te, axis = 2)

y_tr = numpy.reshape(y_tr,(train_samples,output_length,
    vocabulary_size))
y_tr = numpy.argmax(y_tr, axis = 2)

y_te = numpy.reshape(y_te,(test_samples,output_length,
    vocabulary_size))
y_te = numpy.argmax(y_te, axis = 2)

# Calculate training and testing accuracies
t = 0
correct = 0
for i in range(train_samples):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_tr[i,j] == ypred_tr[i,j]:
                correct = correct + 1
print("Training␣accuracy:␣", float(correct)/float(t))

t = 0
correct = 0
```

```
for i in range(test_samples):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_te[i,j] == ypred_te[i,j]:
                correct = correct + 1
print("Testing accuracy: ", float(correct)/float(t))
```

B.3 Branched model data preparation

```
dataX = []
dataY = []
encoding_ratio = int(len(binary)/len(raw_text))
#output_length = int(sequence_length/encoding_ratio)
print("Average integer codeword length: ",encoding_ratio)
#print("Length of each output seqeunce: ", output_length)
output_length = 100
for i in range(0, int(len(binary)/sequence_length), 1):
    seq_in_array = binary[i*sequence_length:(i+1)*sequence_length
       ]
    seq_in = [int(x) for x in seq_in_array]
    seq_out = bitarray(seq_in_array).decode(huffman_dict)
    seq_out_encoded = [char_to_int[x] for x in seq_out]
    seq_out_encoded.extend([len(chars) for _ in range(
       output_length - len(seq_out_encoded))])
    dataY.append(seq_out_encoded)
    dataX.append(seq_in)
```

```
X= numpy.reshape(dataX,(len(dataX),sequence_length))
print("Input shape: ", X.shape)
y = numpy.reshape(dataY,(len(dataY),output_length))
print("Output shape: ", y.shape)
```

B.3.1 Branched DNN model

```
from keras import backend as K
#y = np_utils.to_categorical(y, num_classes = len(chars)+1)
#print("Output shape: ", y.shape)
X_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.3)


inputs = Input(shape=(sequence_length,))
x = Dense(100, activation='relu')(inputs)
#x = Dense((len(chars)+1)*(output_length), activation='relu')(x)
#x = Dense(256, activation='relu')(x)
#x = Dense(100, activation='relu')(x)
#x = Dense(100, activation='relu')(x)



output_name = ["output" + str(i) for i in range(output_length)]
outputdata_name = ["outputdata" + str(i) for i in range(
    output_length)]

for i in range(len(output_name)):
    output_name[i] = Dense(len(chars)+1, activation='softmax')(x)
    outputdata_name[i] = numpy.reshape(y_tr[:,i],(y_tr.shape
        [0],1))
```

```
model = Model(input=inputs, output=[item for item in output_name
    ])
model.compile(optimizer='adagrad',loss='
    sparse_categorical_crossentropy',metrics=['accuracy'])
print("Now fitting the model")
model.fit(X_tr, [item for item in outputdata_name],
    validation_split = 0.2, epochs= 10, batch_size= 64, verbose =
    1)


ypred_tr = numpy.zeros((output_length,X_tr.shape[0],len(chars)+1)
    )
ypred_te = numpy.zeros((output_length,X_te.shape[0],len(chars)+1)
    )
for i in range(output_length):
    layer_output = K.function([model.layers[0].input], [model.
        layers[i+2].output])
    ypred_tr[i] = layer_output([X_tr])[0]
    ypred_te[i] = layer_output([X_te])[0]
ypred_tr = numpy.argmax(ypred_tr, axis = 2)
ypred_te = numpy.argmax(ypred_te, axis = 2)


#t = 0
#correct = 0
#for i in range(y_tr.shape[0]):
#    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
```

```
#                 t = t + 1
#                 if y_tr[i,j] == ypred_tr[i,j]:
#                     correct = correct + 1
#print("Training accuracy: ", float(correct)/float(t))


t = 0
correct = 0
for i in range(y_te.shape[0]):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_te[i,j] == ypred_te[j,i]:
                correct = correct + 1
print("Testing accuracy: ", float(correct)/float(t))
```

B.3.2 Branched CNN model

```
from keras import backend as K
from keras.layers import Activation, Flatten, Merge


X_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.3)
xtr = numpy.reshape(X_tr,(X_tr.shape[0],X_tr.shape[1],1))
xte = numpy.reshape(X_te,(X_te.shape[0],X_te.shape[1],1))


convs = []
kernel_size = [3,5]
inputs = Input(shape=(sequence_length,1))
for i in kernel_size:
```

```
    x = Conv1D(filters = 128, kernel_size = i, strides = 1,
        activation='relu')(inputs)
    x = MaxPooling1D(pool_size=4)(x)
    convs.append(x)
l_merge = Merge(mode= 'concat', concat_axis = 1)(convs)
x = Flatten()(l_merge)

output_name = ["output" + str(i) for i in range(output_length)]
outputdata_name = ["outputdata" + str(i) for i in range(
    output_length)]

for i in range(len(output_name)):
    output_name[i] = Dense(len(chars)+1, activation='softmax')(x)
    outputdata_name[i] = y_tr[:,i]

model = Model(input=inputs, output=[item for item in output_name
    ])
model.compile(optimizer='rmsprop',loss='
    sparse_categorical_crossentropy',metrics=['accuracy'])
print("Now␣fitting␣the␣model")
model.fit(xtr, [item for item in outputdata_name],
    validation_split = 0.2, epochs= 10, batch_size= 32, verbose =
    1)

ypred_tr = numpy.zeros((output_length,X_tr.shape[0],len(chars)+1)
    )
for i in range(output_length):
```

```
    layer_output = K.function([model.layers[0].input], [model.
        layers[i+7].output])
    ypred_tr[i] = layer_output([xtr])[0]
ypred_tr = numpy.argmax(ypred_tr, axis = 2)

ypred_te = numpy.zeros((output_length,X_te.shape[0],len(chars)+1)
    )
for i in range(output_length):
    layer_output = K.function([model.layers[0].input], [model.
        layers[i+7].output])
    ypred_te[i] = layer_output([xte])[0]
ypred_te = numpy.argmax(ypred_te, axis = 2)

t = 0
correct = 0
for i in range(y_tr.shape[0]):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_tr[i,j] == ypred_tr[j,i]:
                correct = correct + 1
print("Training␣accuracy:␣", float(correct)/float(t))

t = 0
correct = 0
for i in range(y_te.shape[0]):
    for j in range(output_length):
```

```
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_te[i,j] == ypred_te[j,i]:
                correct = correct + 1
print("Testing accuracy: ", float(correct)/float(t))
```

APPENDIX C

DNN MODELS FOR TEXT DECOMPRESSION

This appendix lists the code for predicting codeword boundaries from encoded bit sequences [5]

C.1 Data processing

The section of code for processing raw data is same as the previous codes in Appendix 2

```
dataX = []
dataY = []
boundary_output = []
encoding_ratio = int(len(binary)/len(raw_text))
#output_length = int(sequence_length/encoding_ratio)
print("Average integer codeword length: ",encoding_ratio)
#print("Length of each output seqeunce: ", output_length)
output_length = 100

for i in range(0, int(len(binary)/sequence_length), 1):
    seq_in_array = binary[i*sequence_length:(i+1)*sequence_length
        ]
    seq_in = [int(x) for x in seq_in_array]
    seq_out = bitarray(seq_in_array).decode(huffman_dict)
    for item in seq_out:
        boundary_output_array = []
        boundary_output_array.extend([ 1 for _ in range(len(
            huffman_dict[item])-1)])
        boundary_output_array.append(0)
```

```
    boundary_output_array.extend([1 for _ in range(
        sequence_length - len(boundary_output_array))])
    boundary_output.append(boundary_output_array)
    seq_out_encoded = [char_to_int[x] for x in seq_out]
    seq_out_encoded.extend([len(chars) for _ in range(
        output_length - len(seq_out_encoded))])
    dataY.append(seq_out_encoded)
    dataX.append(seq_in)


X= numpy.reshape(dataX,(len(dataX),sequence_length)) #input data
samples = X.shape[0]
print("Input shape: ", X.shape)
y_boundary = numpy.reshape(boundary_output,(samples,
    sequence_length))
y = y_boundary
print("Output shape: ", y.shape)
```

C.1.1 Output concatenation DNN model

```
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
import keras.backend as K


#train test split
X_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.3)
train_samples = X_tr.shape[0]
```

```
test_samples = X_te.shape[0]
#DNN model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim =
    sequence_length))
#model.add(Dropout(0.1))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))

#model.add(Dropout(0.1))
model.add(Dense(y.shape[1], activation='sigmoid'))

#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
#callbacks_list = [checkpoint]

model.fit(X_tr, y_tr, epochs=5, batch_size=16, verbose=1)

reds_te[preds_te>=0.5] = 1
preds_te[preds_te<0.5] = 0
#preds_te_char = preds_te[:,0:output_length*vocabulary_size]
#preds_te_boundary = preds_te[:,:sequence_length]
```

```
#ypred_te_char = numpy.reshape(preds_te_char,(test_samples,
    output_length,vocabulary_size))
#ypred_te_char = numpy.argmax(ypred_te_char, axis = 2)

#y_te_char = y_te[:,0:output_length*vocabulary_size]
#y_te_boundary = y_te[:,:sequence_length]

#y_te_char = numpy.reshape(y_te_char,(test_samples,output_length,
    vocabulary_size))
#y_te_char = numpy.argmax(y_te_char, axis = 2)

t = 0
correct = 0
#for i in range(test_samples):
#    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
#            t = t + 1
#            if y_te_char[i,j] == ypred_te_char[i,j]:
#                correct = correct + 1

for i in range(test_samples):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_te[i,j] == preds_te[i,j]:
                correct = correct + 1
```

```
print("Testing␣accuracy:␣", float(correct)/float(t))
```

C.1.2 Output concatenation CNN model

```
from keras.layers import Dense, Dropout, Activation, Merge
from keras.optimizers import SGD
import keras.backend as K
from keras.layers.convolutional import Conv1D
from keras.layers.pooling import MaxPooling1D


#train test split
X_tr, X_te, y_tr, y_te = train_test_split(X,y,test_size=0.3)
xtr = numpy.reshape(X_tr,(X_tr.shape[0],X_tr.shape[1],1))
xte = numpy.reshape(X_te,(X_te.shape[0],X_te.shape[1],1))

train_samples = X_tr.shape[0]
test_samples = X_te.shape[0]

convs = []
kernel_size = [3,5,7,9]
inputs = Input(shape=(sequence_length,1))
for i in kernel_size:
    x = Conv1D(filters = 128, kernel_size = i, strides = 1,
       activation='relu')(inputs)
    x = MaxPooling1D(pool_size=4)(x)
    convs.append(x)
l_merge = Merge(mode= 'concat', concat_axis = 1)(convs)
x = Flatten()(l_merge)
```

```
x = Dense(784, activation = 'relu')(x)
outputs = Dense(y.shape[1], activation = 'sigmoid')(x)
model = Model(input=inputs, output=outputs)
#DNN model
#model = Sequential()
#model.add(Conv1D(filters = 128, kernel_size = 5, strides = 1,
    activation='relu', padding= 'valid', input_shape=(
    sequence_length,1)))
#model.add(Flatten())
#model.add(Dense(784, activation='relu', input_dim =
    sequence_length))
#model.add(Dropout(0.1))
#model.add(Dense(100, activation='relu'))
#model.add(Dropout(0.1))
#model.add(Dense(y.shape[1], activation='sigmoid'))

#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
callbacks_list = [checkpoint]

model.fit(xtr, y_tr, epochs=5, batch_size=16, verbose=1,
    callbacks = callbacks_list)

preds_te = model.predict(xte)
preds_te[preds_te>=0.5] = 1
```

```
preds_te[preds_te<0.5] = 0
preds_te_char = preds_te[:,0:output_length*vocabulary_size]
preds_te_boundary = preds_te[:,:sequence_length]

ypred_te_char = numpy.reshape(preds_te_char,(test_samples,
   output_length,vocabulary_size))
ypred_te_char = numpy.argmax(ypred_te_char, axis = 2)

y_te_char = y_te[:,0:output_length*vocabulary_size]
y_te_boundary = y_te[:,:sequence_length]

y_te_char = numpy.reshape(y_te_char,(test_samples,output_length,
   vocabulary_size))
y_te_char = numpy.argmax(y_te_char, axis = 2)

t = 0
correct = 0
for i in range(test_samples):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
            t = t + 1
            if y_te_char[i,j] == ypred_te_char[i,j]:
                correct = correct + 1

for i in range(test_samples):
    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
```

```
            t = t + 1
            if y_te_boundary[i,j] == preds_te_boundary[i,j]:
                correct = correct + 1

print("Testing accuracy: ", float(correct)/float(t))
```

APPENDIX D

BIT ERASURE CORRECTION IN COMPRESSED TEXTS

This appendix section lists the code for learning bit to character encoding in presence of erasures. The code for raw data pre-processing is same as before.[5]

D.1 Data processing

```
dataX_1 = []
dataY_1 = []
dataX_2 = []
dataY_2 = []
dataX_3 = []
dataY_3 = []


encoding_ratio = int(len(binary)/len(raw_text))
#output_length = int(sequence_length/encoding_ratio)
print("Average integer codeword length: ",encoding_ratio)
#print("Length of each output seqeunce: ", output_length)
output_length = 100


for i in range(0, int(len(binary)/sequence_length), 1):
    seq_in_array = binary[i*sequence_length:(i+1)*sequence_length
       ]
    seq_in = [int(x) for x in seq_in_array]
    seq_out = bitarray(seq_in_array).decode(huffman_dict)
    seq_out_encoded = [char_to_int[x] for x in seq_out]
```

```
    seq_out_encoded.extend([len(chars) for _ in range(
        output_length - len(seq_out_encoded))])
    dataY_2.append(seq_out_encoded)
    dataY_3.append(seq_in)

for i in range(0, int(len(binary)/sequence_length), 1):
    seq_in_array = binary[i*sequence_length:(i+1)*sequence_length
        ]
    seq_in = [int(x) for x in seq_in_array]
    for j in range(sequence_length):
        if random.random() < erasure_probability:
            seq_in[j] = 2
    dataX_1.append(seq_in)
    seq_out = bitarray(seq_in_array).decode(huffman_dict)
    seq_out_encoded = [char_to_int[x] for x in seq_out]
    seq_out_encoded.extend([len(chars) for _ in range(
        output_length - len(seq_out_encoded))])
    dataY_1.append(seq_out_encoded)

X_1= numpy.reshape(dataX_1,(len(dataX_1),sequence_length)) #input
    data
samples = X_1.shape[0]
print("Input␣shape␣for␣1␣st␣NN:␣", X_1.shape)

output = numpy.reshape(dataY_1,(len(dataY_1),output_length))  #
   output data
yconcat = np_utils.to_categorical(output,num_classes = None)
```

```
y_1 = numpy.reshape(yconcat,(samples,(output_length)*(
    vocabulary_size)))
print("Output shape for 1st NN: ", y_1.shape)

output = numpy.reshape(dataY_2,(len(dataY_2),output_length))  #
    output data
yconcat = np_utils.to_categorical(output,num_classes = None)
y_2 = numpy.reshape(yconcat,(samples,(output_length)*(
    vocabulary_size)))
print("Output shape for 2nd NN: ", y_2.shape)

y_3 = numpy.reshape(dataY_3,(len(dataY_3),output_length))
print("Output shape for 3rd NN: ", y_3.shape)
```

D.2 1st CNN

```
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
import keras.backend as K
from keras.layers.convolutional import Conv1D
from keras.layers.pooling import MaxPooling1D

#train test split
#xtr_1, xte_1, ytr_1, yte_1 = train_test_split(X_1,y_1,test_size
    =0.3)
xtr_1 = numpy.reshape(X_1,(X_1.shape[0],X_1.shape[1],1))
#xte_1 = numpy.reshape(xte_1,(xte_1.shape[0],xte_1.shape[1],1))
train_samples = xtr_1.shape[0]
```

```
#test_samples = xte_1.shape[0]
#DNN model
model1 = Sequential()
model1.add(Conv1D(filters = 256, kernel_size = 5, strides = 1,
    activation='relu', padding= 'valid', input_shape=(
    sequence_length,1)))
model1.add(Flatten())
model1.add(Dense(784, activation='relu', input_dim =
    sequence_length))
#model.add(Dropout(0.1))
#model.add(Dense(100, activation='relu'))
#model.add(Dropout(0.1))
model1.add(Dense(y_1.shape[1], activation='sigmoid'))

#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model1.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
#callbacks_list = [checkpoint]

model1.fit(xtr_1, y_1, epochs=5, batch_size= 256, verbose=1)

#Calculate training and testing predictions
preds_tr_1 = model1.predict(xtr_1)
preds_tr_1[preds_tr_1>=0.5] = 1
preds_tr_1[preds_tr_1<0.5] = 0
#ypred_tr = numpy.reshape(preds_tr_1,(train_samples,output_length
```

```
    ,vocabulary_size))
#ypred_tr = numpy.argmax(ypred_tr, axis = 2)

#preds_te_1 = model1.predict(xte_1)
#preds_te_1[preds_te_1>=0.5] = 1
#preds_te_1[preds_te_1<0.5] = 0
#ypred_te = numpy.reshape(preds_te_1,(test_samples,output_length,
    vocabulary_size))
#ypred_te = numpy.argmax(ypred_te, axis = 2)

#y_tr = numpy.reshape(y_1,(samples,output_length,vocabulary_size)
    )
#y_tr = numpy.argmax(y_tr, axis = 2)

#y_te = numpy.reshape(y_te,(test_samples,output_length,
    vocabulary_size))
#y_te = numpy.argmax(y_te, axis = 2)

# Calculate training and testing accuracies
#t = 0
#correct = 0
#for i in range(samples):
#    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
#            t = t + 1
#            if y_tr[i,j] == ypred_tr[i,j]:
#                correct = correct + 1
```

```
#print("Training accuracy for 1st NN: ", float(correct)/float(t))


#t = 0
#correct = 0
#for i in range(test_samples):
#    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
#            t = t + 1
#            if y_te[i,j] == ypred_te[i,j]:
#                correct = correct + 1
#print("Testing accuracy for 1st NN: ", float(correct)/float(t))
```

D.3 2nd CNN

```
from keras.layers import Dense, Dropout, LSTM


ypred_tr_1 = numpy.reshape(preds_tr_1,(train_samples,
   output_length,vocabulary_size))
ypred_tr_1 = numpy.argmax(ypred_tr_1, axis = 2)
X_2 = ypred_tr_1
xtr_2 = numpy.reshape(X_2,(X_2.shape[0],X_2.shape[1],1))


model2 = Sequential()
model2.add(Conv1D(filters = 128, kernel_size = 5, strides = 1,
   activation='relu', padding= 'valid', input_shape=(X_2.shape
   [1],1)))
model2.add(Flatten())
model2.add(Dense(784, activation='relu', input_dim =
```

```
    sequence_length))
model2.add(Dense(y_2.shape[1], activation='sigmoid'))

#model2.add(LSTM(256, dropout = 0.2, recurrent_dropout = 0.2,
    input_shape=(X_2.shape[1], 1), return_sequences=True))
#model2.add(LSTM(256, dropout= 0.4, recurrent_dropout = 0.1))
#model2.add(Dense(y_2.shape[1], activation='sigmoid'))
#model2.compile(loss='binary_crossentropy', optimizer='RMSprop',
    metrics=['acc'])

#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model2.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
#callbacks_list = [checkpoint]

model2.fit(xtr_2, y_2, epochs=5, batch_size=64, verbose=1)

#Calculate training and testing predictions
preds_tr_2 = model2.predict(xtr_2)
preds_tr_2[preds_tr_2>=0.5] = 1
preds_tr_2[preds_tr_2<0.5] = 0
#ypred_tr_2 = numpy.reshape(preds_tr_2,(samples,output_length,
    vocabulary_size))
#ypred_tr_2 = numpy.argmax(ypred_tr_2, axis = 2)

#preds_te = model.predict(xte)
```

```
#preds_te[preds_te>=0.5] = 1
#preds_te[preds_te<0.5] = 0
#ypred_te = numpy.reshape(preds_te,(test_samples,output_length,
    vocabulary_size))
#ypred_te = numpy.argmax(ypred_te, axis = 2)

#y_tr = numpy.reshape(y_2,(samples,output_length,vocabulary_size)
    )
#y_tr = numpy.argmax(y_tr, axis = 2)

#y_te = numpy.reshape(y_te,(test_samples,output_length,
    vocabulary_size))
#y_te = numpy.argmax(y_te, axis = 2)

# Calculate training and testing accuracies
#t = 0
#correct = 0
#for i in range(samples):
#    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
#            t = t + 1
#            if y_tr[i,j] == ypred_tr[i,j]:
#                correct = correct + 1
#print("Training accuracy for 1st NN: ", float(correct)/float(t))

#t = 0
#correct = 0
```

```
#for i in range(test_samples):
#    for j in range(output_length):
        #if y[i,j]!= y[i,-1]:
#            t = t + 1
#            if y_te[i,j] == ypred_te[i,j]:
#                correct = correct + 1
#print("Testing accuracy for 2nd NN: ", float(correct)/float(t))
```

D.4 3rd CNN

```
ypred_tr_2 = numpy.reshape(preds_tr_2,(samples,output_length,
    vocabulary_size))
ypred_tr_2 = numpy.argmax(ypred_tr_2, axis = 2)
X_3 = ypred_tr_2
xtr_3 = numpy.reshape(X_3,(X_3.shape[0],X_3.shape[1],1))

model3 = Sequential()
model3.add(Conv1D(filters = 16, kernel_size = 5, strides = 1,
    activation='relu', padding= 'valid', input_shape=(
    sequence_length,1)))
model3.add(Flatten())
model3.add(Dense(1000, activation='relu', input_dim =
    sequence_length))
#model.add(Dropout(0.1))
#model3.add(Dense(100, activation='relu'))
#model.add(Dropout(0.1))
model3.add(Dense(y_3.shape[1], activation='sigmoid'))
```

```
#compile and fit the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model3.compile(loss='binary_crossentropy', optimizer=sgd, metrics
    =['accuracy'])
#callbacks_list = [checkpoint]

model3.fit(xtr_3, y_3, epochs=50, batch_size=1, verbose=1)
```

D.5 Accuracy

```
preds_te_3 = model3.predict(xtr_3)
preds_te_3[preds_te_3>=0.5] = 1
preds_te_3[preds_te_3<0.5] = 0


t = 0
correct = 0
for i in range(samples):
    for j in range(sequence_length):
        if X_1[i,j] == 2:
            t = t + 1
            if preds_te_3[i,j] == y_3[i,j]:
                correct = correct + 1

print("Test␣erasure␣accuracy:␣", float(correct)/float(t))
a = model3.evaluate(xtr_3,y_3)[1]
print("Model␣test␣accuracy:␣", a)
```