University of New Orleans

ScholarWorks@UNO

University of New Orleans Theses and Dissertations

Dissertations and Theses

Spring 5-31-2021

Convolutional Neural Networks for Deflate Data Encoding Classification of High Entropy File Fragments

Nehal Ameen
University of New Orleans, New Orleans, nameen@uno.edu

Follow this and additional works at: https://scholarworks.uno.edu/td

Recommended Citation

Ameen, Nehal, "Convolutional Neural Networks for Deflate Data Encoding Classification of High Entropy File Fragments" (2021). *University of New Orleans Theses and Dissertations*. 2853.
https://scholarworks.uno.edu/td/2853

This Thesis-Restricted is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis-Restricted in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis-Restricted has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.

Convolutional Neural Networks for Deflate Data Encoding Classification of High Entropy File Fragments

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science
Cyber Security

by

Nehal Ameen

B.S. Alexandria University, 2012

May 2021

Dedication

To the memory of my grandmother, who despite of the age difference was my best friend and confidant who taught me my values.

To my loving uncle for all his support and encouragement, and for believing in me.

To my adorable partner in crime, my little sister whom I cannot imagine my life without.

To my mom, the strong gorgeous woman who so lovingly dedicated her whole life to her two daughters and always empowered them to go that extra mile.

Acknowledgement

Throughout my work on this thesis, I have received a great deal of support and assistance.

I would like to first thank my advisor, Professor Vassil Roussev whose guidance and support was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to thank Dr. Phani Vadrevu, whose expertise and patient support helped make this work what it is today.

All thanks and appreciation to my friend, former lab partner, and now my professor who served on my thesis defense committee, Dr. Hyunguk Yoo for his wonderful collaboration throughout the years.

In addition, I would like to thank our IT Director and my teacher Matt Toups for helping me throughout his awesome classes, realize my interest in Cyber Security as an undergraduate, and for providing me with the tools that I needed to successfully complete my thesis as a graduate student.

My deepest gratitude to my great mentor and director Pete Persson who supported, motivated, and inspired me through his countless words of encouragement, care, guidance, and advice to cross the finish line.

My wholehearted thanks to Emmanuel Tourniaire whom without his help, support, motivation, and encouragement, I could not have completed this thesis.

My warmest thanks to my colleague and friend Mayra Cervantes who took interest in my work and helped with her technical expertise and advice.

My heartfelt thanks to my friend Sushma Kalle who helped push me forward in times of doubt.

Many thanks to my lovely family and amazing friends who provided stimulating discussions and wise counsel as well as happy distractions to rest my mind outside of my research.

Table of Contents

List of Figures v
List of Tables vi
List of Illustrations vii
List of Code Snippets viii
Abstract ix
Introduction 1
Related Work 4
Problem Statement and Data 8
1- Problem Statement 8
2- Test Data Setup 9
Solution (Implementation) 12
Creating our Datasets 12
Establishing our Heuristics 13
Building our Deep Learning Model 14
Evaluation 22
Conclusions and Future Work 28
References 30
Vita 32

List of Figures

Figure (1): Convolutional Neural Network Structure [17] 15

List of Tables

Table (1): JPEG vs. GZIP Frequency of "FF00" 13
Table (2): GZIP Fragment Encoding Accuracy, True Positives, & True Negatives 13
Table (3): Jpeg, PNG, Gzip (level 9 compression level), Bzip2, Xz, & Zip vs. Gzip (default compression level) Accuracy & Loss 24
Table (4): Docx vs. Gzip Accuracy 25
Table (5): Docx vs. Gzip Loss 25
Table (6): Jpeg, Bzip2, Xz, & Zip vs. PNG Accuracy & Loss 26
Table (8): Xz & Zip vs. Bzip2 Accuracy & Loss 27

List of Illustrations

Illustration (1): Typical Convolutional Neural Network Layers Architecture [16] 16
Illustration (2): A 4-dimensional Word Embedding 17
Illustration (3): Max Pooling 19
Illustration (4): Flattening 20

List of Code Snippets

Code Snippet (1): CNN Architecture 21
Code Snippet (2): Compiling our Model 21
Code Snippet (3): Training & Evaluating our Model 21
Code Snippet (4): Output 21

Abstract

Data reconstruction is significantly improved in terms of speed and accuracy by reliable data encoding fragment classification. To date, work on this problem has been successful with file structures of low entropy that contain sparse data, such as large tables or logs. Classifying compressed, encrypted, and random data that exhibit high entropy is an inherently difficult problem that requires more advanced classification approaches. We explore the ability of convolutional neural networks and word embeddings to classify deflate data encoding of high entropy file fragments after establishing ground truth using controlled datasets. Our model is designed to either successfully classify file fragments that contain hidden patterns and high dimensional features, or to gracefully fail if there are no patterns to be recognized. Our experimental results of the model that we built show high accuracy of 99.82%, 99.73%, and 99.6%, when classifying BZ2, PNG, and GZ against JPEG file fragments, respectively.

Keywords: Cyber Security, Digital Forensics, Data Reconstruction, Deflate Data Encoding, File Fragments, High Entropy, Data Science, Big Data Analytics, Machine Learning, Deep Learning, Convolutional Neural Networks, Word Embeddings

Chapter 1

Introduction

Reliable data encoding (fragment) classification is important for efficient data reconstruction in terms of speed and accuracy, as evidence could be found in deleted hidden fragments where file carving technologies are usually applied to reconstruct files from these fragments. Highly accurate classification relies on full file extension, the magic number, or the metadata of files, which only works when the metadata is found within the fragments extracted from a storage medium. Usually, that is not the case in real life, which is why a more practical approach is needed to classify data encodings of file fragments accurately.

File type and data encoding are two different concepts that are often confused. Starting with the basic notion of a file before differentiating between the two concepts, a file is a sequence of bytes that is stored by a file system under a user-specified name. *A data encoding is a set of rules for mapping pieces of data to sequences of bits*. Such an encoding is *minimal*, if it is not possible to reduce the rule set and still produce meaningful data encodings. The same piece of information can be represented in different ways using different encodings. For example, a plain text document could be represented in ASCII for editing, and in compressed form for storage/transmission. Once encoded, the resulting bit stream can serve as the source for further (recursive) encodings, e.g., a base64- encoded JPEG image [5].

A file type is a set of rules for utilizing (sets of) primitive data encodings to serialize digital artifacts. Unlike data encodings, file types can have very loose, ambiguous, and extensible sets of rules, especially for complex file types. Consider the breakdown of recursively embedded MS Office objects found inside a set of ~20,000 MS Office files. A Word document may contain a PowerPoint presentation, which in turn may contain an Excel spreadsheet, which may contain OLE objects, and so on [5].

File fragment classification is the process of mapping a sample chunk of data, such as a disk block, to a specific type of data encoding [5]. Small pieces of data (fragments) are usually found on disk blocks or network packets. On the low end, file system space is usually allocated in fixed-sized blocks of 512, 1024, 2048, or 4096 bytes; on modern systems, 4KB is the most common while higher allocation are also used.

Data encoding classification is a complicated problem because there are many kinds of file types—from simple primitive types such as a block of ASCII text or a JPEG file, to complex container files such as an Adobe Acrobat File (PDF), to archive files such as TAR and ZIP that can themselves contain many other files (and even other archives). Given a contiguous piece of data (a fragment), the classification is the encoding format of the file from which the fragment was taken [4].

The general problem of file fragment classification is formulated in two separate questions, 1) What is the primitive data format of the fragment? 2) Is the fragment part of a compound file structure? Our research focuses on the first question. Given a fragment, a classification method should be able to detect evidence of known primitive encoding formats, such as jpeg. For primitive types we can measure the size of a fragment, which usually falls between 256 and 4096 bytes vs. correct classification. Separately, we can measure the effectiveness of classification with and without header information. Classification methods should also be able to state 'I don't know' when the data exceeds their abilities [4].

In this work, we explore a new approach to classify file fragments of high entropy. We assume a realistic scenario where there are no headers, metadata, or magic numbers, thereby making the classification more challenging. This research explores the ability of deep learning techniques, specifically convolutional neural networks and word embeddings using TensorFlow [1] to solve this problem.

The two most common uses of Convolutional Neural Networks (CNNs) are 1) image classification, 2) time series forecasting. Neural Networks in general mimic the way our nerve cells communicate through interconnected neurons. CNNs have a similar architecture, but what makes them unique from other neural networks is the convolutional operation that applies filters to every part of the previous input in order to extract patterns and features maps, which makes them more sensitive to patterns that could be otherwise hidden. Since high entropy file fragments may have hidden patterns that can be difficult to extract, a convolutional neural network seemed to be the most suitable solution to this problem.

Another concept that we combined with our solution is Word Embeddings. Word embedding is the collective name for a set of language modeling and feature learning techniques in Natural Language Processing (NLP) where words or phrases from the vocabulary are mapped to vectors of real numbers to learn the position of a word. Learning the position of a word – in our case a byte – in a learned vector space, will help our model learn different patterns in the different data encodings. It can be learned as part of a deep learning model. Since word embeddings have been shown to boost the performance in NLP tasks such as syntactic parsing and sentiment analysis [6], and our classification problem could fall under syntactic parsing can, we added an embedding layer to our convolutional neural network to help with our pattern extraction, which improved our results significantly.

The case for specialized approaches to file fragment classification has been made as shown later in chapter 2. However, it is not scalable. Our approach combines a more generic approach with the specialized approach to enhance accuracy and scalability. The specialized part of our approach treats primitive data format and compound file structure separately to accurately classify deflate encoded file fragments such as JPEG, PNG, Gzip, Zip, Bzip2, and Docx. The more generic piece of the puzzle is our deep learning model.

Also, clean, structured and content-controlled datasets are not readily available to perform accurate unbiased classifications. The first step that we took in our research was establishing ground truth using a clean controlled structure and data content to leave no room for mislabeling due to similar formatting. We used our knowledge of some of the existing prevalent patterns in JPEG fragments to build heuristics to set a baseline on which we can measure the performance of our classification model. We used the different block sizes as the fragment sizes for our classifications, starting with 256 bytes and going up until 4096 bytes to see how different fragment sizes affect the accuracy of our classifier.

In this research, we are using TensorFlow to build our convolutional neural network model and files from the public Gov Docs **Error! Reference source not found.**[2] and the msx-13 corpus [3] to create our datasets. TensorFlow includes tf.keras, a high-level neural network API that provides useful abstractions to reduce boilerplate and make TensorFlow easier to use without sacrificing flexibility and performance [1]. We also built our environment in Colab, a Google Research product, which allows developers to write and execute Python code through their browser. Google Colab is an excellent tool for deep learning tasks. It is a hosted Jupyter notebook that requires no setup and has an excellent free version, which gives free access to Google computing resources such as GPUs and TPUs [7].

We evaluate our model using four different file types, JPEG, PNG, GZ, ZIP, BZIP2, and DOCX files. DOCX files are zip files that consist of deflate encoded files/components (almost entirely in xml), and embedded media content that is stored in its original (compressed) encoding. Generally, small objects (deflate) are good news as the beginning/end have readily identifiable markers and can save us deeper analysis [5].

Our model is able to classify JPEG file fragments against gz file fragments with 99.60% accuracy, which met our heuristic performance that achieved around 99.2% accuracy. This result has not been achieved using other methods or other deep neural network implementations, which shows that our model is actually capable of recognizing any patterns in file fragments. Our model is also capable of saying “I don’t know” if a classification is impossible due to the absence of patterns in randomized or encrypted compressed data. Our results show a 57.63% accuracy when it tries to classify zip fragments against gzip fragments, which is the expected result given that both formats use the same RFC 1951 compression format.

Chapter 2

Related Work

Content-based file fragment classification algorithms such as extracting the N-gram, Shannon entropy, Hamming weight and statistical regularities of bytes have been proposed for file fragments of low and medium entropy file structures, such as: large tables or logs and text or code. In similar schemes, traditional machine learning techniques are deployed to improve the performance of these classification algorithms. However, for high entropy files such as compressed files (e.g., .zip files or. jpg files), different classification techniques would be more efficient [5].

(Poisel, Rybnicek, & Tjoa, 2013) identified several categories of data fragment classifiers. Their taxonomy divided them into the following main-classes: signature-based approaches, statistical approaches, computational intelligence-based approaches, approaches considering the context, and other approaches.

Signature-based approaches use byte-sequences for the identification of unknown file fragments by matching typical and well-known byte sequences. A wide-spread application area in the context of digital forensics is to determine header and footer fragments by file signatures which are often referred to as magic number. Statistical approaches use quantitative analysis techniques to identify fragments of given file types. Statistical properties such as the mean value, variance, binary frequency distribution (BFD), or the rate of change (ROC) are determined from fragments contained in reference data sets to obtain a model for each data type. The actual classification is then carried out by comparing the fragments in question to the precalculated model.

For computational intelligence approaches, they explained that the goal is to transform data into information after learning from a collection of given data. For data fragment and type classification, strong classifiers have to be trained. They further refine this class into supervised (if the training set consists of labeled data) and unsupervised (if patterns and structures are derived from unlabeled data) approaches. Both supervised and unsupervised machine learning algorithms are used to meet the goal of correct classification of data fragments and file type classification.

As for context-considering approaches, information gained from meta-data extracted from other fragments or the transport medium is used. Such approaches can provide additional information necessary for the correct classification. The category “other approaches” contains techniques which cannot be assigned to one of the other categories. A special sub-class of class are combining approaches [11]. Our work can be categorized under “computational intelligence – supervised approach” since we train our CNN model using labeled datasets.

(Duffy, 2014) investigates the role of ML in file fragment classification as a proof of concept. Because the research in this field is limited, and no specialized algorithms existed to solve the fragment classification problem, they chose SVMs because they are generally a good candidate for any classification problem. Naive Bayes was used to see how well they could model this problem like a text classification problem, treating each of the bytes as if they were randomly occurring words in some text stream. LDA was used just as a method of comparison to the other two, as it is also a generally strong classification algorithm[9].

While they performed their testing on file fragments, for parameter estimation they trained each model on entire files. This was done with the goal in mind of creating models that could recognize whole files, and then assume that fragments given to the model will have the same distributions (in general) as whole files with the corresponding type. This method performed fairly well at allowing them to classify the random testing fragments. SVM obtained an accuracy of 75.03%, Multinomial Naïve Bayes achieved 47.9%, and LDA resulted in a 69.01% accuracy.

The author mentioned a few issues with their dataset that may skew the results. The first is that the extensions of some files in the dataset are inaccurate, and while this does not give cause to disregard the results wholly, it does mean that there is some amount of error in the approximations. The other issue with the dataset is the number of file types trained on. The classification performed here is far from optimal and lacks intelligence to distinguish well between multiple different types of the same format. For example, the classifier often misclassified .html files as .py files. This is in part due to the issues with the dataset as noted above [9].

(Xu, et al., 2014) explored the feasibility of the idea that whether the type of a file fragment can be detected by description of its corresponding grayscale image using an image classification method. Considering a grayscale image of 256 gray-level, every 8 bits in the data can be viewed as a pixel in a grayscale image. Every grayscale image has its width and height. Therefore, it is necessary to reshape “pixels” into a 2D array to make them constitute an image instead of a line. All the images were converted from 1024 bytes fragments with 32 pixels (32 bytes in file fragments) in width. These images can be classified into correct types easily by human eyes. Therefore, if computers can view pictures like human beings, file fragments may be classified based on their corresponding descriptions of grayscale images.

After obtaining the descriptions of file fragments, the vectors were put into different classification algorithms to find the best classification algorithm which fits for file fragment classification. In this work, 4 commonly used classification algorithms were evaluated: 1) KNN, 2) Naïve Bayes, 3) SVM, and 4) Decision Tree. The classification experiments were conducted in WEKA and ten-fold cross validation was used to evaluate their approach.

Their result showed that KNN was fast to build the model required for classification because of its simplicity. The performance of KNN was the best in most cases. Two models based on file-unbiased and type-unbiased were proposed to verify the validity of the proposed method. The best average classification accuracy acquired is 39.7% with K being 1 in 9 dimensions in file-unbiased model. While in type-unbiased model, the best average classification accuracy is 54.7% with K being 7 in 9 dimensions [12]. The classifier here performed well on low entropy file fragments, but is not applicable to high entropy file fragments, which reflected findings in previous work.

After that, (Chen & Liao, 2018) added deep learning to the grayscale image conversion approach to extract more hidden features and therefore improve the accuracy of classification. Their proposed CNN model achieved 70.9% accuracy. Some of the grayscale images have obvious texture features different from the others, while some of them look quite similar, such as the grayscale images of DOCX, GIF, GZ, JPG, and PNG. JPG files use the lossy compression algorithm, while GIF files are based on the LZW algorithm. DOCX, GZ, and PNG are produced using Phil Katz's Deflate compression algorithm. Deflate is a lossless data compression algorithm that uses both the LZ77 algorithm and Huffman Coding, which explains the stark similarities between them. They can be easily confused because they are either embedded or compressed high entropy files.

Due to the ability of CNNs to extract high-dimensional features, PNG and GZ; both are high entropy compressed files, could be separated to some degree. However, many files such as, PNG and GZ were misclassified. Since GZ is a compressed file and PPT is a composite file type, they may embed different types of file fragments which can skew the results. Additionally, this paper did not optimize the data, the distributions of different types of files are not the same in the GovDocs datasets that were used, so if the number of files of a certain type is significantly less than others, it will affect the accuracy of the final classification results [8]. There was no consistency in the files selected and the file type was not distinguished from the file extension in this work, which led to a 70.9% overall accuracy and that is not enough when taking into account the huge amounts of data that need to be classified.

(Hiester, 2018) explored the use of neural networks as universal models for classifying file fragments, focusing on the lossless feature representation, with fragments' bits as direct input, and its use of feedforward, recurrent, and convolutional networks as classifiers. The recurrent networks achieved 98% accuracy in distinguishing 4 file types, suggesting that this approach may be capable of yielding models with sufficient performance for practical applications.

Due to the study's exploratory nature, the models were not directly evaluated in a practical setting; rather, easily reproducible experiments were performed to attempt to answer the initial question of whether this approach is worthwhile to pursue further. Additionally, the experiments tested classification of fragments of homogeneous file types as an idealized case, rather than using a realistic set of types. A random fragment of 512 bytes was selected from each file of a size no less than 1,024 bytes, using the bits as features, hence, the lossless representation. Their research was focused on CSV, XML, JPG, and GIF files where their feed-forward neural network achieved 77% accuracy, their recurrent neural network achieved 98% accuracy, and their convolutional neural network achieved 73% accuracy [10].

The previous work has a number of methodological problems, which our work strives to address. These begin with the fundamental problems of not distinguishing between "file type" and "file extension", not distinguishing between the primitive data format of the fragment and the fragment being part of a compound file structure, and the inconsistency of files selected. For our research, we took the lessons learned from the research that has been conducted over the past decade in this area and we focused on building a more generic specialized classification approach that focused on primitive data formats, specifically the high entropy deflate encoded files jpeg, png, zip, Gzip, Bzip2, and docx. We chose a computational intelligence approach to build a specialized strong classifier using supervised learning. Before training our model, we established ground truth using a clean controlled structure and data content to leave no room for mislabeling due to similar formatting. We also optimized our datasets by using equal numbers of samples from each file type for training and testing with a consistent ratio between our training and testing datasets, around 80% to 20% respectively for all our experiments.

Chapter 3

Problem Statement and Data

1- Problem Statement

The file fragment classification problem refers to the problem of taking a file fragment and automatically detecting the file type. This is an important problem in digital forensics, particularly for carving digital files from disks. The problem with file fragment classification is that it is quite complicated due to the sheer size of the search space and the different kinds of file types—from simple primitive types to complex container files. In order to perform a proper file fragment classification, two questions need to be answered, the first is, what the primitive data format of the fragment is and the second is whether the fragment is part of a compound file structure. Moreover, the definition of the file type can be quite vague, and often file types are only characterized by their header information [4].

Research has been done on file fragment classification for a long time using a variety of different approaches, such as: signature-based, statistical, computational intelligence-based, approaches considering the context, and other approaches like combining several approaches (Taxonomy). In the computational intelligence based or machine learning description of the problem, each file type is thought to be a category (class) and certain features that are thought to characterize the file fragment are extracted. Then, supervised machine learning approaches are used to predict the category label for each test instance. Some of the methods also incorporate unsupervised machine learning approaches [11].

Some file fragments are easier to classify than others, depending on the file structure and the fitness of the classification approach followed. For example, Jpeg header recognition is relatively easy to accomplish – the header has a variable length record structure in which synchronization markers are followed by the length of the field. Thus, some simple 'header hopping' can reliably identify the header [5].

JPEG body recognition is also not difficult to accomplish as the encoding uses byte stuffing that results in the 16-bit hexadecimal FF00 occurring on average every 191 bytes [4]. Placed next to a high-entropy sample with a different encoding, e.g., deflate, this feature should stick out rather prominently [5]. However, classifying high entropy fragments such as compressed files against each other is not as easy. So far, simple classifiers that provide a quick and general classification have been implemented. However, for high entropy files such as compressed files (e.g., .zip files or. gzip files), different classification techniques would be more efficient.

The lack of classification approaches that treat primitive data format and compound file structure separately, to accurately classify deflate encoded file fragments such as JPEG, PNG, Gzip, Zip, and Docx is a problem that we need to overcome. Also, clean, structure and content-controlled datasets are not readily available to perform accurate unbiased classifications. Additionally, the majority of the available classification tools are not designed to handle large amounts of data and take several hours if not days to execute.

In this work we investigate a more specialized approach that focuses on primitive data formats, specifically, Huffman encoded file fragments and test the ability of deep learning to recognize any hidden patterns. We try to find the optimal deep learning architecture, model, and hyper parameter values for such classifications and how much each fragment size can reveal, using clean controlled datasets that we created. Our tool produces expected results within a few minutes.

2- Test Data Setup

File compression takes advantage of redundancy or patterns to "abbreviate" the contents of the file in such a way to take up less space yet maintain the ability to reconstruct a full version of the original when needed [13].

In our experiments, we investigate the classification of Gzip file fragments against JPEG, PNG, Zip, and Docx file fragments with different groupings. Docx files are zip files that consist of deflate encoded files/components (almost entirely in xml), and embedded media content that is stored in its original (compressed) encoding [5].

The jpeg format has some distinctive data format encoding features, which are helpful with respect to fragment classification. Detecting the JPEG header is separate from the detection of the encoded image. The header has a simple record structure where the beginning of each record is announced by the presence of a marker—a 16-bit number in the 0xFFC0 to 0xFFFE range, which is followed by a 16-bit number describing the length of the record. We are focused on the JPEG body of the image as the true problem of significance. It is fairly straightforward to identify compressed/encrypted data using some basic entropy measurements. The true task is to differentiate among different compressed streams. Apart from zlib, most compressed formats do have some synchronization information. We mentioned earlier that, in the body of the image, jpeg encoders stuff a 0x00 byte after every 0xFF. In addition to that, there are a few more legal markers that may appear—mostly in the 0xD0 to 0xDB range [4].

The zlib data format is employed by zip and gz files. The zlib/deflate encoding (RFC, 1950/1951) is entirely focused on storage efficiency and contains the absolute minimal amount of metadata necessary for decoding. It consists of a sequence of compressed blocks, each one comprised of:

3-bit header - The first bit indicates whether this is the last block in the sequence; the following two bits define how the data is coded: raw (uncompressed), static Huffman, or dynamic Huffman. In practice, dynamic Huffman is present 99.5% of the time [4].

Huffman tables - These describe the Huffman code books used in a particular block. The Huffman encoding scheme assigns codes to characters such that the length of the code depends on the relative frequency or weight of the corresponding character. It takes advantage of the disparity between frequencies and uses less storage for the frequently occurring characters at the expense of having to use more storage for each of the rarer characters. Huffman is an example of a variable-length encoding— some characters may only require 2 or 3 bits and other characters may require 7, 10, or 12 bits. The savings from not having to use a full 8 bits for the most common characters makes up for having to use more than 8 bits for the rare characters and the overall effect is that the file almost always requires less space [4][13].

Compressed data - The table is followed by a stream of variable-length Huffman codes that represent the content of the block. One of the codes is reserved for marking the end of the block. As soon as the end-of-block code is read from the stream, the next bit is the beginning of the following block header, there is no break in the bit stream between blocks, and there are no synchronization markers of any kind. The end-of-block code depends on the coding table, so it varies from block to block. The upshot is that absent sanity checking, where a deflate decoder can sometimes "decode" even random data. The statistical variation of the coded stream is quite uniformly random [4][5].

To ensure that we build a clean setup, we control the structure and content of the data instead of being blind in the wild. To control artifacts in JPEG file fragments, we remove all header metadata (using ExifTool) [14] and trim the beginning of the resulting file to remove readily recognizable strings in the file header. We treat PNG files the same way. For Gzip file fragments, we merge a large number of HTML files and compress them using the Gzip command in Linux with the default compression level "-6". As for Zip file fragments, we Zip-compress a folder containing a large number of HTML files. This is to ensure that Gzip and Zip file fragments do not contain any embedded images or objects that could skew the results one way or another.

To make sure we do the same with Docx file fragments, we remove any /media folders contained within the compressed Docx folder and we group the Docx files by their sizes to create our datasets. One group contains Docx files that are greater than or equal 16 KB and less than 32 KB, the second group contains Docx files that are greater than or equal 32 KB and less than 64 KB, the third group contains Docx files that are greater than or equal 64 KB and less than 128 KB, and finally, the fourth group contains Docx files that are 128 KB or larger.

We also optimized our datasets by using equal numbers of samples from each file type for training and testing with a consistent ratio between our training and testing datasets, around 80% to 20% respectively for all our experiments. We ran our experiments using different fragment sizes starting by 256 bytes and going up until 4096 bytes to see how different fragment sizes affect the accuracy of our classifier. Each sample resembled a fragment, and each byte resembled a feature. While running our experiments, we optimized our hyperparameters each time until we achieved sensible results.

In the following section we go over the methodology that we used to create our datasets from the aforementioned file types.

Chapter 4

Solution (Implementation)

Creating our Datasets

To obtain the required data, we downloaded Jpeg and HTML files from the public Gov Docs corpora. We converted some JPEG files to PNG files, due to the lack of PNG files in the Gov Docs corpora [2] and downloaded Docx files from the msx-13 corpus [3]. We used the HTML files to create our zip and gz files. For the zip files, we added all the HTML files to a folder and zip compressed this folder. As for the gz files, we merged all the HTML files into one large file and compressed this file using the gzip command in Linux with the compression level set to default (-6).

Then, to clean the data and establish content control we trimmed the header and removed metadata from the body of Jpeg files using ExifTool. We also removed all /media folders from the Docx files, and we excluded any files that were less than 4KB in size. For further experimentation, we made copies of the docx files that were grouped into different sizes, 1) between 16 – 32 KB, 2) between 32 – 128 KB, 3) 128 KB or larger. All that was to ensure that we only have stand-alone primitive files.

We wrote a Python[15] program to create different labeled datasets from these files. The basic idea of this program is that it converts each file into its decimal value, then adds a ".0" to each decimal value to change the values into float values to then be processed by TensorFlow. The reason behind this data conversion is, as explained above, there are certain known hexadecimal patterns in some file types, such as the occurrence of the hexadecimal FF00 every 191 bytes in a jpeg body [4]. The decimal value of the byte FF is 255 and 0 for the byte 00, which with our conversion becomes 255.0,0.0. Each byte represents one feature in our dataset.

The idea is to create training and testing datasets for each file type with a training to testing ratio 0.8:0.2. Each dataset is created in a CSV file where each row of bytes (features) represents a fragment. We wanted our model to train and test different fragment sizes, and because we have a fixed number of files with fixed sizes, there is a trade-off between the fragment size and the number of samples that we can use, the larger the fragment size is, the less the number of samples that we have. Another aspect that we have to take into account is that we want to have equal number of samples for each file type in both training and testing datasets, so we have to base our calculations on the file type with the minimum sum of file sizes. We then create our training and testing files with fragment sizes 256, 512, 1024, 4096 bytes for each type. The reason behind selecting these particular fragment sizes is explained in chapter 1.

We then select the fragment size and the file types that we want to classify. Based on the order of the type selection, each type is labeled using an integer to represent this type starting by 0 and incrementing by 1 for each new type. Then the selected training files are concatenated together with the added “label” column, the testing files are treated similarly. We only experimented with binary classifications for this thesis work.

Establishing our Heuristics

We used our knowledge of some of the existing prevalent patterns in JPEG fragments to build simple heuristics to establish a baseline for the expected results based on the prevalence of the FF00 pattern, so we can measure the performance of our classification model. As shown below, we used a fragment size of 1024 bytes and 25,000 training samples for each of the types Jpeg and Gzip and 10,000 testing samples for each. We then found the minimum and maximum occurrence of the pattern FF00 among all the fragments of each type.

As expected, the number showed a huge difference between the max occurrence in jpeg, 84 times compared to only 2 times in gzip. We then found the number of samples that had 0, 1, 2, and 3 occurrences of the pattern within each type, to find the optimal number of occurrences (frequency) to use as a basis for our heuristic. We found out that if we go with a rule that is as simple as “if frequency == 0, then the fragment belongs to a gzip file” achieved approximately 97.5% accuracy, with only 2% false positives. Based on that analysis we decided that we should not accept any accuracy that is less than 97.5% for this particular classification, if not higher.

Table (1): JPEG vs. GZIP Frequency of “FF00”

Frequency of "FF00" by dataset

	JPEG				GZIP			
1024 bytes	**Training**	**%**	**Testing**	**%**	**Training**	**%**	**Testing**	**%**
min	0		0		0		0	
max	84		97		2		2	
0	620	2.48	618	6.18	24328	97.31	9866	98.66
1	1514	6.06	1220	12.20	646	2.58	132	1.32
2	2315	9.26	1698	16.98	26	0.10	2	0.02
3	2835	11.34	1573	15.73	0	0.00	0	0.00
Samples per set	25,000		10,000		25,000		10,000	

Table (2): GZIP Fragment Encoding Accuracy, True Positives, & True Negatives

Accuracy from True Positives & True Negatives

GZIP	**Accuracy**	**True Positive**	**False Positive**
if freq == 0, then GZIP	~ 97.50%	~ 97.00%	~ 2.00%
if freq >= 2, then JPEG	~ 95.70%	~ 99.90%	~ 8.50%

Building our Deep Learning Model

As we explained in chapter 2, we wanted to use deep learning to solve this problem. We built our model in TensorFlow as it allows creating custom layers for your neural network. Many machine learning models are expressible as the composition and stacking of relatively simple layers, and TensorFlow provides both, a set of many common layers as a well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers [1].

We chose Google Colab for our environment due to its ease of use and the availability of TPU runtime environments which significantly reduced the runtime of the simulations. Colab is a Google Research product, which allows developers to write and execute Python code through their browser. It is an excellent tool for deep learning tasks. It is a hosted Jupyter notebook that requires no setup and has an excellent free version, which gives free access to Google computing resources such as GPUs and TPUs [7].

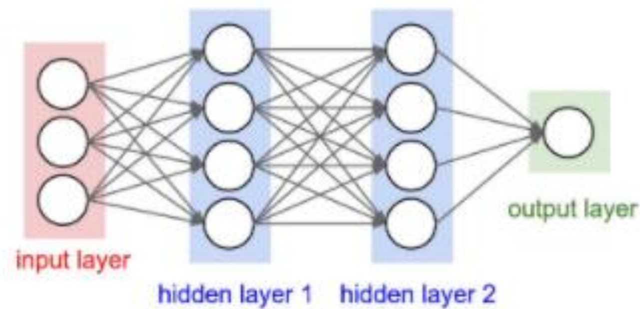
A model is the relationship between features and the label. A good machine learning approach determines the model for you. If you feed enough representative examples into the right machine learning model type, the program will figure out the relationships. There are many types of models and picking a good one needs experience and a lot of experimentation [1]. A huge part of our research was determining the model to train.

Our experiments were structured as follows:

1- Import and parse training and testing datasets.
2- Build the model.
3- Train the model using training datasets.
4- Evaluate the model's effectiveness using testing datasets.
5- Use the trained model to make predictions.

Neural networks can find complex relationships between features and the label. It is a highly structured graph, organized into one or more hidden layers. Each hidden layer consists of one or more neurons. The diagram below shows a visualization of how nodes or neurons of the input and output layers are connected through hidden layers that work together to learn the different features that the Neural Network needs to know to be able to distinguish one class from the other. There are several categories of neural networks and the model we chose was Convolutional Neural Networks with an Embedding layer. Figure (1) shows a generic Convolutional Neural Network Structure.

Figure (1): Convolutional Neural Network Structure [17]

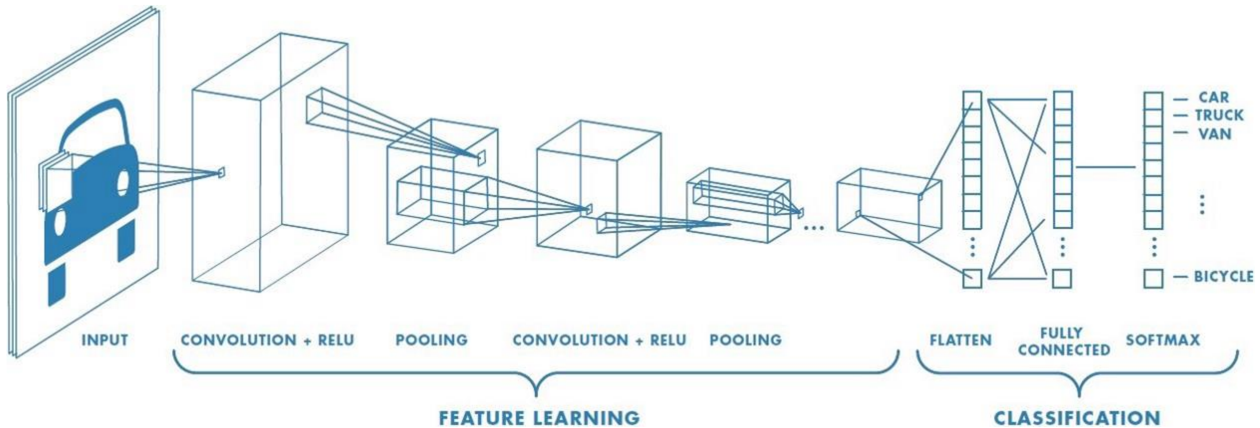


Convolutional Neural Networks are the leading algorithms in today's world which are used to solve the Computer vision problems such as: 1) Image Classification tasks, 2) Facial Recognition tasks, 3) Object Detection, 4) Pattern Detection, and 5) Natural Language Processing.

One of the most popular research in this area was the development of LeNet-5 by LeCunn and co. in 1997. This was one of the first Convolutional Neural Networks (CNN) that was deployed in banks for reading cheques in real-time. It is said that the LeNet-5 read over a million cheques. Although there were other algorithms Like Support Vector machines which were close to the accuracy of the LeNet-5, it was argued that the CNN speed of computation was exponentially faster than other algorithms.

A one-dimensional CNN is a CNN model that has a ***convolutional hidden layer*** that operates over a 1D sequence. This is followed by perhaps a second convolutional layer in some cases, such as very long input sequences, then a pooling layer whose job it is to distill the output of the convolutional layer to the most salient elements. The convolutional and ***pooling layers*** are followed by a ***dense fully connected layer*** that interprets the features extracted by the convolutional part of the model. A ***flatten layer*** is used between the convolutional layers and the dense layer to reduce the feature maps to a single one-dimensional vector as shown in illustration (1) below that showcases a typical CNN Architecture [1][6][16][17].

Illustration (1): Typical Convolutional Neural Network Layers Architecture [16]



We used the Keras Sequential API to add our neural network layers. A “Sequential” model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. It can be created incrementally via the add( ) method. All layers in Keras need to know the shape of their inputs in order to create their weights. Initially when a layer is created, it has no weights. It creates its weights the first time it is called on an input, since the shape of the weights depends on the shape of the inputs. So, when a Sequential model is instantiated without an input shape, it is not built and thus, has no weights. The weights are only created when the model first sees some input data. Once a Sequential model is built, every layer will have an input and output attribute. These attributes can be used to do neat things, like quickly creating a model that extracts the outputs of all intermediate layers in a Sequential model [1].

Our convolutional base consists of:

1- Embedding Layer [1][6]

Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding. An embedding is a dense vector of floating-point values (the length of the vector is a parameter you specify). Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer). It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024 dimensions when working with large datasets. A higher dimensional embedding can capture fine-grained relationships between words but takes more data to learn. Conceptually, it involves a mathematical embedding from a space with many dimensions per word to a continuous vector space with a much lower dimension. One of the methods to generate this mapping is neural networks.

Illustration (2) below is a diagram for a word embedding. Each word is represented as a 4-dimensional vector of floating-point values. Another way to think of an embedding is as a "lookup table". After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.

Illustration (2): A 4-dimensional Word Embedding

car	1.2	-0.1	4.7	3.1
truck	0.4	3.5	-0.7	0.6
van	0.3	2.2	0.3	0.4

Word embeddings provide a dense representation of words and their relative meanings, which is an improvement over the more traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast, and a given word or document would be represented by a large vector comprised mostly of zero values.

Instead, in an embedding, words are represented by dense vectors, where a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. That position of a word in the learned vector space is referred to as its embedding. Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data and can be considered as a class of approaches for representing words and documents using a dense vector representation.

The output of the *Embedding* layer is a 2D vector with one embedding for each word in the input sequence of words (input document), which can be understood as a lookup table that maps from integer indices (which stand for specific words) to dense vectors (their embeddings). The dimensionality (or width) of the embedding is a parameter you can experiment with to see what works well for your problem, much in the same way you would experiment with the number of neurons in a Dense layer.

When you create an Embedding layer, the weights for the embedding are randomly initialized (just like any other layer). During training, they are gradually adjusted via backpropagation. Once trained, the learned word embeddings will roughly encode similarities between words (as they were learned for the specific problem your model is trained on). If you pass an integer to an embedding layer, the result replaces each integer with the vector from the embedding table.

Learning the position of a word – in our case a byte – in a learned vector space, will help our model learn different patterns in the different data encodings. Keras offers an embedding layer that can be used for neural networks on text data. It requires that the input data be integer encoded, so that each word is represented by a unique integer. In our case, each byte is represented by a unique float value, which is automatically cast to an integer within the layer. The Embedding layer is initialized with random weights and will learn an embedding for all of the words (bytes) in the training dataset. It is a flexible layer that can be used in a variety of ways, such as:

- It can be used alone to learn a word embedding that can be saved and used in another model later.
- *It can be used as part of a deep learning model where the embedding is learned along with the model itself, which is the case in our experiment.*
- It can be used to load a pre-trained word embedding model, as a type of transfer learning.

The Embedding layer is defined as the first hidden layer of a network. It must specify 3 arguments:

- **input_dim**: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0 – 10, then the size of the vocabulary would be 11 words. In our case, the data is encoded to values between 0 – 255, so the size would be 256.
- **output_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. The optimal number is determined by experimenting with different values. We tested different values for our problem and 10 performed well.
- **input_length**: This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all of your input documents are comprised of 1000 words, this would be 1000. This represents the number of features, in our case, the fragment size. We used 256, 512, 1024, and 4096 for our input_length values.

2- Convolution + Activation Layer [1][6][17]

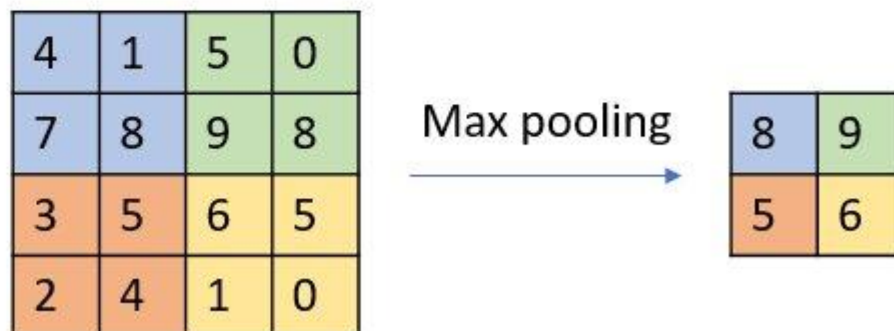
Convolution is an interesting operation that works by taking a 'Feature map' or say a 3x3 filter and applying it on every part of the input features. Here, applying means an arithmetic operation where the result of the operation is stored as a value for the next layer. This operation is repeated across the input image by "Convolving" the filter. A Convolution layer can have a number of Feature Maps (or Filters) in each layer and so can produce as many outputs as possible. In our model, we used 20 filters. The size of each filter is configured using the **kernel_size** parameter.

After applying the convolutional function, a non-linearity is added to the output. It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function). Typically, this is done by the Rectified Linear Unit (Relu) Activation function. You can think of this as passing only the positive values to the output while changing the negative values to 0.

3- Pooling Layer [1][6][17]

Pooling is an operation which has 2 main impacts, 1) It reduces the dimensions of the feature maps, so lesser parameters are faster to compute in following layers. Hence, it is also known as a down-sampling layer, 2) It highlights the importance of the features. There are a few pooling operations which are popular: Average pooling, Max pooling and Sum Pooling. Out of these max pooling is the most widely used operation, so we chose it for our model. Below is a visualization of how it works. Similar to the kernel_size, the size of the max pooling window is an integer that's configured using the **pool_size** parameter. We used 32 for our model. Illustration (3) shows how Max Pooling works.

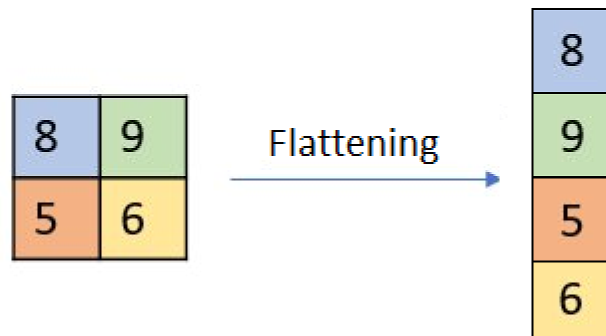
Illustration (3): Max Pooling



4- Flatten Layer [1][6]

The flatten layer basically takes the current pooling layer output and it converts it into the format which is required for the Fully connected layer. The fully connected layer is an artificial neural network in itself and requires a specific input. Illustration (4) shows how the Flattening Layer works.

Illustration (4): Flattening



5- Dense (Fully Connected) Layer [1][6][17]

The initial convolution layers help in detecting low level features. When we pass these again into additional convolutional layers, higher level features are detected. The fully connected layer is the final piece of the puzzle. It takes the high-level feature maps as the input and decides what the output category would be. This is basically a multi-level Perceptron network that identifies which weights are more likely to contribute to which outputs. This is done when we train the model with a lot of samples, it is able to decide which attributes associate more with which categories. It is fully connected as every neuron in the previous layer is connected to every neuron in the next layer. The activation function of our dense layer is the **Sigmoid Function** because it returns a value between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output, as in our model. Since the probability of anything exists only between the range of 0 and 1, sigmoid is the correct activation function.

This is a code snippet to show our CNN architecture explained above:

Code Snippet (1): CNN Architecture

```
1 model = models.Sequential()
2 model.add(tf.keras.layers.Embedding(256, 10, input_length=4096))
3 model.add(layers.Conv1D(filters=20, kernel_size=2, activation='relu', name="conv1"))
4 model.add(layers.MaxPooling1D(pool_size=32))
5 model.add(layers.Flatten())
6 model.add(layers.Dense(1, activation='sigmoid'))
```

After building our neural network we need to compile it using the following code. It uses Adam, which is a momentum-based optimizer. The loss function used is binary_crossentropy, which is usually the optimizer of choice for binary classification problems that give output in the form of probability. The metric we used is accuracy.

Code Snippet (2): Compiling our Model

```
1 model.compile(optimizer='adam',
2               loss='binary_crossentropy',
3               metrics=['accuracy'])
```

The model is then trained on the training set for 10 epochs, then evaluated for the test set to check the accuracy, as shown in the code and output snippets below.

Code Snippet (3): Training & Evaluating our Model

```
1 fitted_model = model.fit(train_dataset, epochs=10)
2 eval_model = model.evaluate(test_dataset)
```

Code Snippet (4): Output

```
Epoch 1/10
442/442 [==============================] - 86s 124ms/step - loss: 0.4648 - accuracy: 0.7612
Epoch 2/10
442/442 [==============================] - 70s 123ms/step - loss: 0.0331 - accuracy: 0.9943
Epoch 3/10
442/442 [==============================] - 68s 118ms/step - loss: 0.0151 - accuracy: 0.9977
Epoch 4/10
442/442 [==============================] - 68s 119ms/step - loss: 0.0082 - accuracy: 0.9987
Epoch 5/10
442/442 [==============================] - 68s 118ms/step - loss: 0.0033 - accuracy: 0.9997
Epoch 6/10
442/442 [==============================] - 68s 118ms/step - loss: 0.0031 - accuracy: 0.9998
Epoch 7/10
442/442 [==============================] - 68s 119ms/step - loss: 0.0013 - accuracy: 1.0000
Epoch 8/10
442/442 [==============================] - 78s 141ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 9/10
442/442 [==============================] - 74s 125ms/step - loss: 6.3195e-04 - accuracy: 1.0000
Epoch 10/10
442/442 [==============================] - 69s 120ms/step - loss: 5.2319e-04 - accuracy: 1.0000
111/111 [==============================] - 39s 72ms/step - loss: 0.0363 - accuracy: 0.9960
```

Chapter 5

Evaluation

We now have our CNN model, ready to be trained on our datasets that we created. For the sake of this research, we focused on the high entropy Huffman encoded data. We used binary classifiers to classify pure jpeg, png, gz, zip, bz2, xz, and docx fragments. We ran our simulations for each fragment size for each classification separately. Each simulation took approximately between 15 minutes to 30 minutes to complete.

To evaluate the performance of our model, we introduce here two important parameters [1][6]:

- $$Accuracy = \frac{True\ Positives + True\ Negatives}{\text{True Positives + True Negatives + False Positives + False Negatives}} \quad (1)$$

- *Loss:* Our binary classification problem can be posed as: "is the fragment a gzip fragment?" or, "what is the probability of the fragment being gzip?" In this setting, gzip fragments belong to the positive class (Yes, they are gzip fragments), while other fragments belong to the negative class (No, they are not gzip fragments). When we fit a model to perform this classification, it predicts a probability of being gzip to each one of our samples. A loss function evaluates how good or bad the predicted probabilities are. For a binary classification like ours, the typical loss function is the Binary Crossentropy function (BCE), which is used to determine the error (aka "the loss") between the output of our algorithms and the given target value. In layman's terms, the loss function expresses how far off the mark our computed output is.

  Since we are trying to compute a loss, we need to penalize bad predictions. If the probability associated with the true class is 1.0, we need its loss to be zero. Conversely, if that probability is low, say, 0.01, we need its loss to be huge, which is calculated by taking the negative log of the probability. The binary cross-entropy is computed using the following equation:

  $$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log\big(p(y_i)\big) + (1 - y_i) \cdot \log\,(1 - p(y_i)) \quad (2)$$

  $\boldsymbol{y}$ is the **label** (1 for gzip fragments and 0 for each of the other fragments) and $\boldsymbol{p(y)}$ is the predicted probability of the fragment being gzip for all **N** samples. What this formula tells us is that for each gzip fragment ($\boldsymbol{y}$= 1), it adds $\boldsymbol{log(p(y))}$ to the loss, that is the log probability of it being gzip. Conversely, it adds $\boldsymbol{log(1\text{-}p(y))}$, that is, the log probability of it being jpeg, zip, or docx for each other fragment ($\boldsymbol{y}$= 0).

We started with our baseline classification, which is classifying jpeg and gz fragments, since we know what results to expect. Using a fragment size of 256 bytes did not really help the network learn any patterns as the number of features used was too low. As we increased the fragment size, the accuracy improved gradually, starting at 57.91% for 256 bytes and going up to 99.60% when we used 4096 bytes. Conversely, the loss decreased from 3.2213 to 0.0004. We show the distributions of accuracy and loss values for our classifications in **Table (3)** below. After that, we wanted to see the effect of the embedding layer on the accuracy of the classification jpg vs. gz. We tried classifying them after removing the embedding layer from our network which reduced the accuracy significantly from 99.6% to 70.91% as shown in **Table (3)** as well.

After successfully classifying jpeg and gz fragments, we wanted to test if our model is just picking on the "FF00" pattern or if it is picking on a different pattern that is not obvious to the human eye. So, we omitted each occurrence of "FF00" to get rid of this specific pattern and, the test accuracy started at 50.02% at 256 bytes, but surprisingly, it went up to 99.52% at 4096 bytes. We noticed that the accuracy was much less than the results we had without omitting "FF00" at 512 and 1024 bytes, but the values still reveal that there is probably a different pattern that the model is picking up on. That could be some legal markers that may appear—mostly in the 0xD0 to 0xDB range in the jpeg fragments [4]. It is worth digging deeper and understanding what these results could be revealing about the structure of the jpeg fragments, but that is out of the scope of this work.

Classifying zip and gz fragments started at an expected accuracy of 50% at 256 bytes and stayed at 50% all the way through until it got to 57.63% at 4096 bytes, which is explained by the fact that both zip and gz fragments are of high entropy, there are hardly any patterns to recognize. We also tried classifying gz fragments with the default compression level against gz fragments with the highest compression level to increase the entropy, but the accuracy was still 50%. The loss distributions are shown in **Table (3).**

For our docx and gz classification, we tried two different approaches to building our datasets. The first approach was to group all docx files that were higher than 4 KB together and pick up-to 5 random samples from each file, which started at a surprising accuracy of 100% at 256 bytes and stayed at a 100% for 512 and 1024 bytes, then significantly went down to 50.27% at 4096 bytes as shown in **Table (4)**. **Table (5)** shows the loss starting at a very low value of 0.1246 and increasing to 2.9956. The second approach was grouping the files by sizes between 16 to 32KB, 32 to 128KB, and 128KB & larger, and also picking up to 5 random samples from each file in each group, leaving us with datasets that are separated, not only by the fragment size, but also by the docx size grouping. Accuracy results are shown in **Table (5).**

Below are our “Accuracy by fragment size & datasets” and “Loss by fragment size & datasets” tables, showing the results that we got from our CNN model for each of our classifications. Each table is divided into 4 sections for our 4 different fragment sizes that we trained and tested, each of the fragment sections is sub-divided into training and testing subsections. Then each type that was classified has the accuracy or loss recorded for the last epoch of the training and recorded for the testing (evaluation), followed by the number of samples used in each dataset.

Table (3): Jpeg, PNG, Gzip (level 9 compression level), Bzip2, Xz, & Zip vs. Gzip (default compression level) Accuracy & Loss

Classification Task	256		512		1024		4096	
	Training	**Testing**	**Training**	**Testing**	**Training**	**Testing**	**Training**	**Testing**
	Accuracy by fragment size & dataset (GZ vs. ALL)							
JPEG	99.15%	57.91%	99.44%	81.17%	99.83%	97.02%	100.00%	99.60%
JPEG “FF00” omitted	98.66%	50.02%	99.02%	68.26%	99.52%	83.52%	100.00%	99.52%
JPEG w/o Embedding							84.00%	70.91%
Bzip2	97.90%	50.26%	97.69%	53.40%	97.64%	81.33%	98.40%	95.47%
PNG	98.26%	50.01%	98.28%	59.98%	98.39%	76.46%	97.77%	79.58%
Zip	96.71%	50.00%	93.96%	50.00%	90.07%	50.00%	73.73%	57.63%
Xz	96.61%	50.00%	94.33%	50.00%	90.19%	50.00%	75.83%	56.84%
Gzip -9	96.72%	50.00%	93.87%	50.00%	88.21%	50.00%	46.30%	50.01%
	Loss by fragment size & dataset							
JPEG	0.0314	3.2213	0.0179	0.8701	0.0052	0.1018	0.0005	0.0004
JPEG w/o Embedding							0.5092	1.1114
Bzip2	0.0779	4.0311	0.0650	2.0640	0.0634	0.4739	0.0649	0.1257
PNG	0.0673	4.6678	0.0540	2.5475	0.0536	1.3210	0.0747	1.0726
Zip	0.1212	4.5419	0.1790	2.6664	0.2164	1.3597	0.5347	0.6927
Xz	0.1253	4.7295	0.1572	2.8696	0.2159	1.3890	0.5106	0.7639
Gzip -9	0.1260	4.8047	0.1795	2.7974	0.2459	1.5505	0.6949	0.6935
Samples per set	113,024	28,256	56,512	14,128	28,256	7,064	7,064	1,766

Table (4): Docx vs. Gzip Accuracy

Docx (without /media)	Accuracy by fragment size & dataset (Docx vs. GZ) 256		512		1024		4096	
	Training	**Testing**	**Training**	**Testing**	**Training**	**Testing**	**Training**	**Testing**
4 – 128+ KB	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	50.27%
Samples per set	860	215	860	215	848	212	371	93
16 – 32 KB	100.00%	53.01%	100.00%	83.99%	100.00%	50.10%	100.00%	50.12%
Samples per set	964	241	964	241	964	241	834	209
32 – 128 KB	100.00%	50.12%	100.00%	50.12%	100.00%	50.12%	100.00%	50.12%
Samples per set	804	201	804	201	804	201	804	201
128+ KB	100.00%	50.65%	100.00%	50.65%	100.00%	50.65%	100.00%	49.35%
Samples per set	156	39	156	39	156	39	156	39

Table (5): Docx vs. Gzip Loss

Docx (without /media)	Loss by fragment size & dataset (Docx vs. GZ) 256		512		1024		4096	
	Training	**Testing**	**Training**	**Testing**	**Training**	**Testing**	**Training**	**Testing**
4 – 128+ KB	0.0008	0.1246	0.0005	0.0271	0.0003	0.1908	0.0007	2.9956
Samples per set	860	215	860	215	848	212	371	93
128+ KB	0.1803	0.7634	0.1196	0.6603	0.0387	0.7846	0.0191	0.7327
Samples per set	156	39	156	39	156	39	156	39

Table (6): Jpeg, Bzip2, Xz, & Zip vs. PNG Accuracy & Loss

Classification Task	256		512		1024		4096	
	Training	Testing	Training	Testing	Training	Testing	Training	Testing
Accuracy by fragment size & dataset (PNG vs. ALL)								
JPEG	99.56%	51.30%	99.70%	73.34%	99.90%	94.21%	100.00%	99.73%
Bzip2	99.06%	50.01%	98.88%	50.22%	99.05%	87.17%	99.54%	99.14%
Xz	99.01%	50.00%	98.64%	50.00%	98.25%	50.29%	97.49%	98.67%
Zip	99.03%	50.00%	98.47%	50.01%	98.00%	54.82%	96.78%	97.14%
Loss by fragment size & dataset								
JPEG	0.019	4.5163	0.0108	1.3755	0.0030	0.2364	0.0000	0.0058
Bzip2	0.0418	5.2402	0.0386	3.5419	0.0275	0.3062	0.0141	0.0256
Xz	0.0418	5.3504	0.0497	3.7353	0.0433	1.6211	0.0693	0.0551
Zip	0.0444	5.4883	0.0553	3.7410	0.0491	1.0722	0.0857	0.1009
Samples per set	211,232	52,816	105,616	26,408	52,808	13,204	13,202	3,301

Table (7): Xz, Bzip2, & Zip vs. JPEG Accuracy & Loss

Classification Task	256		512		1024		4096	
	Training	Testing	Training	Testing	Training	Testing	Training	Testing
Accuracy by fragment size & dataset (JPG vs. ALL)								
Xz	99.44%	54.43%	99.68%	76.71%	99.89%	97.44%	100.00%	100.00%
Bzip2	99.41%	50.89%	99.57%	61.39%	99.84%	90.78%	100.00%	99.82%
Zip	99.44%	52.19%	99.56%	71.26%	99.84%	93.45%	99.99%	99.76%
Loss by fragment size & dataset								
Xz	0.0250	3.7994	0.0124	1.2623	0.0030	0.0912	0.0000	0.0001
Bzip2	0.0285	4.5272	0.0175	2.3269	0.0056	0.2946	0.0001	0.0044
Zip	0.0256	4.2616	0.0170	1.6862	0.0054	0.5610	0.0004	0.0052
Samples per set	211,232	52,816	105,616	26,408	52,808	13,204	13,202	3,301

Table (8): Xz & Zip vs. Bzip2 Accuracy & Loss

Classification Task	256		512		1024		4096	
	Accuracy by fragment size & dataset (BZ2 vs. ALL)							
	Training	Testing	Training	Testing	Training	Testing	Training	Testing
Xz	98.68%	50.00%	98.42%	50.00%	98.61%	64.73%	99.25%	97.29%
Zip	98.58%	50.00%	98.27%	50.00%	98.31%	58.86%	98.95%	95.97%
	Loss by fragment size & dataset							
Xz	0.0564	5.5756	0.0543	3.5189	0.0400	0.8877	0.0250	0.0687
Zip	0.0609	5.625	0.0625	4.1661	0.0506	1.1671	0.0322	0.1044
Samples per set	211,232	52,816	105,616	26,408	52,808	13,204	13,202	3,301

Chapter 6

Conclusions and Future Work

Data encoding classification simply is the process of mapping a sequence of bytes from a file (fragment) to a specific type of data encoding. Reliable data encoding classification of file fragments improves the speed and accuracy of data reconstruction significantly. In general, classifying data encodings of low to medium entropy file fragments is much easier than classifying data encodings of high entropy file fragments. So far, all machine learning approaches that have been designed to classify file fragments have had methodological issues that we tackled in this thesis work.

While designing our solution, we took into account the conceptual difference between the notions of file type and data encoding, as well as the difference between primitive data format and compound file structure, making our classification more efficient. We used our knowledge of the anatomy of different data encodings to build our heuristics using ground truth that we established through controlled and clean datasets that we prepared to help evaluate our results.

We followed a computational intelligence-based approach using a convolutional neural network with a word embedding layer to build a more generic and scalable, yet specialized classification approach that focused on Huffman encoded (high entropy) file fragments. Our classifier has the ability to either successfully classify file fragments that contain hidden patterns and high dimensional features, or to say, “I don’t know” and gracefully fail if there are no patterns to be recognized.

Our solution has 2 main components, the first is our dataset builder, that can create any datasets needed for training and testing, with any fragment size, any number of samples, and any training to testing ratio desired. The second component is the classifier itself; it can be fed any training and testing datasets and it will output training and testing accuracy results within a few minutes. We ran all our classifications through the same fragment sizes (256, 512, 1024, and 4096 bytes) for consistency and to better understand how much can be extracted from each fragment size. We were also able to run at least four different classifier instances simultaneously, which gave us the ability to obtain results for at least four different classifications within 15 minutes only.

Our binary classifiers achieved 99.6% accuracy when classifying JPEG and GZ fragments, between 97.14 and 99.73% when classifying PNG against ZIP, XZ, BZIP2 and JPEG fragments, and between 99.76% and 100% when classifying JPEG against ZIP, BZIP2, and XZ fragments.

Our main research contribution is that we have demonstrated a new fragment classification approach that, unlike prior work, is *both generic and exhibits very high accuracy*, making it suitable for practical application at scale. Further, unlike most prior work, our results are based on a large, realistic, but also carefully curated datasets to ensure that ground truth is known, rather than assumed (based on file extension). By excluding extraneous metadata, such as those contained in file header, we have constructed the most difficult test case, which gives us confidence that the results would be reproducible in the real world. Finally, our approach has been able to classify different variations of the same basic (high entropy) data encoding, *deflate*, which is another first.

Our future work would be building a multi-classifier using our binary classifiers to build decision trees as well as exploring other multi-classification approaches. We will also include cross validation in our steps and use larger fragment sizes, up to 16 KB to classify other deflate/ Huffman encoded file fragments that were not addressed in this research. Additionally, we plan to use LIME (Local Interpretable Model-Agnostic Explanations) to explain the ability of our neural network to classify JPEG fragments even after omitting the “FF00” pattern.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Garfinkel, Farrell, Roussev and Dinolt, Bringing Science to Digital Forensics with Standardized Forensic Corpora, DFRWS 2009, Montreal, Canada

[3] Roussev, V. (n.d.). *the msx-13 corpus*. Retrieved from Vassil Roussev: http://roussev.net/msx-13/msx-13.html

[4] Roussev, V., & Garfinkel, S. L. (2009). File Fragment Classification - The Case for Specialized Approaches. New Orleans: University of New Orleans.

[5] Roussev, V., & Quates, C. (2013). File fragment encoding classificationd - An empirical approach . *DFRWS*.

[6] Goodfellow, I., Bengio, Y., & Courville, A. (2015). *Deep Learning.*

[7] Bisong E. (2019) Google Colaboratory. In: Building Machine Learning and Deep Learning Models on Google Cloud Platform. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-4470-8_7

[8] Chen, Q., & Liao, Q. (2018). File Fragment Classification Using Grayscale Image Conversion and Deep Learning in Digital Forensics.

[9] Duffy, A. (2014). CarveML: application of machine. Stanford University.

[10] Hiester, L. (2018). File Fragment Classification Using Neural Networks with Lossless Representations. East Tennessee State University.

[11] Poisel, R., Rybnicek, M., & Tjoa, S. (2013). Taxonomy of Data Fragment Classification.

[12] Xu, T., Xu, M., Ren, Y., Xu, J., Zhang, H., & Zheng, N. (2014). A File Fragment Classification Method Based on Grayscale Image. College of Computer, Hangzhou Dianzi University, Hangzhou, China.

[13] Zelenski, J., Huffman, K., Schwarz, K., & Stepp, M. (2012). Huffman Encoding and Data Compression.

[14] Harvey, P. (2016). ExifTool. Retrieved from https://exiftool.org/

[15] Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.

[16]Introduction to Convolutional Neural Networks (Stanford University, 2018)

[17]Introduction to Convolutional Neural Networks (Stanford University, Spring 2021)

Vita

Nehal Ameen is an M.S. candidate in the Computer Science department at UNO, where she worked as a Graduate Research Assistant under Dr. Roussev's supervision. She was awarded a Bachelor of Science in Management Information Systems at Alexandria University in Egypt. Nehal is currently working as an Emerging Technology Product & Systems Analyst at Entergy, where she started as a Robotic Process Automation Developer. As a Computer Scientist, her interests include Machine Learning Engineering and Cyber Security, and finding innovative approaches to enhance Cyber Security using AI. She has worked on multiple projects focused on Big Data Analytics, Digital Forensics, and Software Reverse Engineering, including working on a portable visual field system prototype that innovates a non-invasive method for visual field-testing using VR goggles, as a Visiting Scholar at Tulane University. She co-authored "CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC" and earned credits from the National Cyber Security and Communications Integration Center for reporting three vulnerabilities in Modicon's PLC while working as a Research Assistant in the CyPhy lab at UNO.