MALWARE DETECTION USING TRANSFORMERS-BASED MODEL GPT-2

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS INSTITUTE
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

NAZENİN ŞAHİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CYBER SECURITY

NOVEMBER 2021

Approval of the thesis:

MALWARE DETECTION USING TRANSFORMERS-BASED MODEL GPT-2

submitted by **NAZENİN ŞAHİN** in partial fulfillment of the requirements for the degree of **Master of Science in Cyber Security Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, **Graduate School of Informatics** ______________

Assist. Prof. Dr. Cihangir Tezcan
Head of Department, **Cyber Security** ______________

Assoc. Prof. Dr. Cengiz Acartürk
Supervisor, **Cognitive Science Dept., METU** ______________

Examining Committee Members:

Assist. Prof. Dr. Cihangir Tezcan
Cyber Security Dept., METU ______________

Assoc. Prof. Dr. Cengiz Acartürk
Cognitive Sciences Dept., METU ______________

Assist. Prof. Dr. İlker Özçelik
Software Engineering Dept., OGU ______________

Date: 17.11.2021

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Nazenin ŞAHİN

Signature :

ABSTRACT

MALWARE DETECTION USING TRANSFORMERS-BASED MODEL GPT-2

Şahin, Nazenin
M.S., Department of Cyber Security
Supervisor: Assoc. Prof. Dr. Cengiz Acartürk

November 2021, 62 pages

The variety of malicious content, besides its complexity, has significantly impacted end-users of the Information and Communication Technologies (ICT). To mitigate the effect of malicious content, automated machine learning techniques have been developed to proactively defend the user systems against malware. Transformers, a category of attention-based deep learning techniques, have recently been shown to be effective in solving various malware problems by mainly employing Natural Language Processing (NLP) methods. In the present study, we propose a Transformers architecture to detect malicious software automatically. We present models based on GPT-2 (Generative Pre-trained Transformer 2), which performs assembly code obtained from a static analysis on PE (Portable Executable) files. We generated a pre-trained model to capture various characteristics of both malicious and benign assembly codes. That improves the model's detection performance. Moreover, we created a binary classification model that used preprocessed features to characterize existing malicious and benign code pieces. The resulting binary classification model distinguishes between those code pieces by recognizing novel malware or benign assembly codes. Finally, we used GPT -2's pre-trained model to improve detection accuracy. The experiments showed that a fine-tuned pre-trained model and GPT-2's pre-trained model led to accuracy values up to 85.4% and 78.3%, respectively.

Keywords: Malware Detection, Static Analysis,Opcode, Transformers, GPT-2, NLP

ÖZ

TRANSFORMATÖR MODEL GPT-2 KULLANARAK ZARARLI YAZILIM TESPİTİ

Şahin, Nazenin
Yüksek Lisans, Siber Güvenlik Bölümü Bölümü
Tez Yöneticisi: Doç. Dr. Cengiz Acartürk

Kasım 2021 , 62 sayfa

Zararlı içeriğin çeşitliliği, karmaşıklığının yanı sıra Bilgi ve İletişim Teknolojilerinin (BİT) son kullanıcılarını önemli ölçüde etkilemiştir. Zararlı içeriğin etkisini azaltmak, kullanıcı sistemlerini zararlı yazılımlara karşı proaktif olarak savunmak için otomatikleştirilmiş makine öğrenme teknikleri geliştirildi. Dikkate dayalı derin öğrenme tekniklerinin bir kategorisi olan Transformers'ın, son zamanlarda, Doğal Dil İşleme (NLP) yöntemlerini kullanarak, çeşitli zararlı yazılım sorunlarını çözmede etkili olduğunu gösterilmiştir. Bu çalışmada, zararlı yazılımları otomatik olarak tespit etmek için bir Transformers mimarisinin kullanılmasını öneriyoruz. PE (Portable Executable) dosyaları üzerinde statik analizden elde edilen montaj kodları ile GPT-2'ye (Generative Pre-trained Transformer 2) dayalı modelleri besliyoruz. Hem zararlı hem de zararsız montaj kodlarının çeşitli özelliklerini yakalamak için önceden eğitilmiş bir model oluşturduk. Yakalanan bu özellikler modelinin tespit performansını iyileştirir. Ayrıca, mevcut kötü amaçlı ve zararsız kod parçalarını karakterize etmek için önceden işlenmiş özellikleri kullanan bir dil modeli oluşturduk. Böylece, ortaya çıkan dil modeli, yeni zararlı veya zararsız yazılımların derleme kodlarını tanıyarak bu kod parçaları arasında ayrım yapar. Ek olarak, daha iyi tespit doğruluğu elde etmek için GPT -2'nin önceden eğitilmiş modelini de kullandık. Deneyler, bizim önceden eğitilmiş modelimiz ve GPT-2'nin önceden eğitilmiş modeli ile ince ayar yapıldığında, tespit modelinin sırasıyla %85,4 ve %78,3'e doğruluk değerlerine ulaştığını göstermiştir.

Anahtar Kelimeler: Zararlı Yazılım Tespiti, Statik Analiz, Opcode, Transformatörler, GPT-2, NLP

To my mom, Filiz

ACKNOWLEDGMENTS

First, I would like to thank Assoc. Prof. Dr. Cengiz Acartürk, my supervisor, for his priceless guidance, continuous support, encouragement, and most of all, his patience throughout the entire process.

I would like to thank the members of our malware analysis research group, Melih Şırlancı, and Deniz Demirci, for their support in the scope of this study. I also would like to thank Mr. Demirci, especially for his encouragement.

Lastly, I truly appreciate my administrator Col. Adnan Gürbüz, who encouraged me to start this journey. I would like to thank my sister Figen, my brothers Nedim, Kutluay, Mutlu, and my best friends Barış and İdil for their support throughout every moment of my life.

TABLE OF CONTENTS

CHAPTERS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligent
ANN	Artificial Neural Network
API	Application Programming Interface
BFE	Byte-Pair Encoding
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
GPT-2	Generative Pretrained Transformer 2
LM	Language Model
LSTM	Long Short-Term Memory
ML	Machine Learning
MLM	Masked Language Model
MLP	Multilayer Perceptron
NLP	Natural Language Processing
NN	Neural Network
PE	Portable Executable
RNN	Recurrent Neural Network

CHAPTER 1

INTRODUCTION

The evaluation of Information and Communication Technologies (ICT) has significantly impacted the variety of malicious content besides its complexity in mitigation methods. Malicious software spread rapidly in networks by the increasing connectivity of new devices such as end-user computers and servers, cloud systems, smartphones, and IoT devices. The exponential increase in the number of malware also leads to substantial economic loss. According to Deep Instinct reports, in 2020, the variety of malware increased by more than three times, and ransomware increased by more than four times compared to 2019 (Shimon, 2021).

Every day, over 350,000 new malware and potentially unwanted applications (PUA) are registered by the AV-TEST[1] Institute. Approximately 114 million new malicious programs were developed in 2020, 84.4% targeting Windows operating systems. Nevertheless, this expansion of malware is not bound to specific operating systems. More specifically, according to McAfee reports, MacOS malware expanded more than four times in the third quarter of 2020, largely due to a specific malware (aka. malicious software), namely the EvilQuest ransomware. A Windows operating system ransomware, driven by the Cryptodefense engine, grew in volume by 69% in the third quarter of 2020. In addition, malware developers attack mobile operating systems and mobile apps due to the significant increase (118% from Q3 to Q4 in 2020, (Samani, 2021)) in the mobile devices market. Linux operating system was not an exception though with a lesser impact. Overall, Linux malware increased 6% from Q3 to Q4 in 2020. The economic impact of malware has increased to a considerable degree, too. For instance, Sodinokibi, a well-known ransomware group, claims that they gained $123 million in 2020 out of ransomware exploitations (Lemos, 2021). The economic impact of malware is also evident in other reports. For instance, IBM reported that the average cost of a data breach was $3.86 million in 2020 (Burmester, 2020). A common method of spread in malware attacks is phishing. As of January 2021, Google registered more than 2 million phishing sites, showing a 27% annual increase (Rosenthal, 2020). More generally, malicious software aims to exploit vulnerabilities in ICT systems, gain unauthorized access to valuable information assets, or render them unusable and ask for ransom, as in ransomware.

The expansion of malicious software on the Internet requires more and more efficient mitigation methods. Those are not only established upon signature-based anti-malware tools but also anti-malware that employs machine learning methods. A significant issue with the signature-based systems is that the databases of signatures

[1] avtest.org: https://www.av-test.org/en/statistics/malware/ (retrieved on 19 Jun 2020)

cannot keep up with the growth of new malware. A review of the literature reveals numerous examples of the integration of learning mechanisms into malware detection methodologies. In the present study, we contribute to malware detection by employing a language model, specifically GPT-based pre-trained and binary classification model. The idea behind the models is that suspicious files exhibit similar coding patterns in terms of their inherent malicious content and targets.

There are two approaches, namely static and dynamic, to investigate suspicious files for malware analysis. In dynamic approaches, the malware analysts gain information by executing the suspected files and tracing the execution flow to examine the function calls. The function calls inform which function is called and which operations are performed, so the examiners decide whether files are malicious or not. On the other hand, in static approaches, the analysts investigate without executing the files. The disassembler/debugger tools, such as *objdump*[2] and IDA Pro[3] have been used by analysts to disassemble files for obtaining import functions, strings, and assembly codes that may facilitate identifying the attacks (Gandotra et al., 2014). As in classical malware analysis, pre-trained models and binary classification model also extract coding patterns from byte codes[4], such as byte-sequence n-grams, or assembly codes[5], such as assembly instructions[6], (opcodes[7] or operands[8]), opcode sequences. The models determine whether the patterns are malicious or not using machine learning techniques and deep learning techniques (Acarturk et al., 2021).

Machine learning techniques, in particular deep learning (DL) techniques are used to detect malware on various fronts, not only conducting binary classification of softwares as benign or malicious but also classifying malware into known types such as virus, worm, and trojan. In our study, DL models learn representations (i.e., embeddings) from assembly instructions by encoding opcodes and operands. Then they identify the distance between the embeddings of two instructions to compute their similarity. The smaller the distance, the more similar functions are to each other. Deep learning methods are based on defining a language model on assembly instructions using various methods, such as the Generative Pre-trained Transformer 2 (GPT-2) (Radford et al., 2019). The assembly language model is trained on various traces to detect each assembly instruction effect in its context. Next, the binary classification model transfers the learned knowledge from the assembly language model, viz. the pre-trained model. This method is fine-tuned by the pre-trained model. The binary classification model achieves to match syntactic and semantic similarities of assembly instructions with the pre-trained model. Thus, the binary classification model classifies softwares as benign or malicious and uses GPT-2 architecture by the similarities, namely binary classification. The transformer (Wolf et al., 2020) processes data on short-text, i.e., sentence-level tasks such as paraphrase detection and sentiment analysis, or on short document texts such as reading comprehension and automatic summarization of news articles, and defines a new state-of-the-art with an attention mechanism that provides global dependencies between input and output. In summary,

[2] *objdump*: http://objdump.com (retrieved on: 20.05.2020)

[3] IDA Pro: https://hex-rays.com/ida-pro/ (retrieved on: 20.05.2020)

[4] Byte code: Computer source code.

[5] Assembly code: Human readable code of executables.

[6] Assembly Instructions: Each row of assembly code consisting of opcode and operand.

[7] Opcode: opcode is a single instruction that the CPU execute.

[8] Operand: operand is a data or memory address used to execute that opcode.

the goal of a Natural Language Processing (NLP) DL model, within the context of malware analysis, is the quantification of syntactic and semantic characteristics of software code. The syntactic and semantic similarities are used for the analysis with real-world security usages, such as vulnerability detection with assembly instructions (Brumley et al., 2008), exploiting generation with C code (Avgerinos et al., 2011), tracing malware lineage with assembler instructions (Bayer et al., 2009). We have to deal with softwares at assembly instructions during the matching structural or semantic similarities of instructions for malware detection. However, since the programs are compiled with various compiler optimizations, they are run in different instruction set architectures, making it challenging to establish similarities between assembly instructions (Pei et al., 2021). Therefore, we used Windows Portable Executable (PE) files which display malware codes as Intel ×86 assembly instructions, and use disassembler *objdump*. Moreover, we used the Hugging Face library to implement models. Hugging Face has an extendable framework and an open-source NLP library used by global services, including Bing, Apple, and Monzo. [9]

1.1 Motivation and Problem Definition

We analyzed the malware detection problem owing to its major impact on information systems. Language model and text classification approaches have the potential to provide solutions to this problem, so we adapted them into the malware detection context. Our decoder-based context-aware network Generative Pre-trained Transformer 2 (GPT-2) language model retains contextual characteristics of syntactic and semantic similarities from the natural language sentences (i.e., short-text) perspective due to the attention mechanism layers. It is also called the transformers-based language model. The model provides global dependency on inputs and outputs with attention mechanisms. We used the next word prediction perspective of GPT-2, a decoder-based network for binary classification of malware.

We fed into our models with statically collected assembly instructions to do binary classification. The statistical analysis identifies malicious applications among benign applications. As there is no need to activate the malware by executing the code to capture the features, statistical approaches are also less expensive in terms of resources and time. We disassembled software files that were not obfuscated to obtain assembly codes using static analysis tools. Like the GPT-2 transformers-based model, deep learning methods allow us to extract relevant features even from complex instructions of assembly codes. The neural networks also automatically handle feature extraction instead of manual feature extraction in machine learning. Transformers-based models also clearly exhibit remarkable results in various state-of-the-art Natural Language Processing (Tay et al., 2020) and computer vision tasks (Naseer et al., 2021).

We developed a language model on the syntactic and semantic representation of assembly instructions from malicious and benign executable files. This model transferred its knowledge about the structure of assembly codes, namely transfer base learning or pre-trained model, to the binary classification model. Hence, the binary classification model increased discovery capability with the help of the pre-trained

[9] HuggingFace: https://huggingface.co/ (retieved on 22.05.2020)

model.

1.2 Research Questions and Approach

The research questions of this study are presented as follows. Firstly, our study investigates the models that aim to represent the semantics of malware's assembly code, specifically focusing on GPT-2. Secondly, it targets conducting comparative evaluations between alternative models while keeping efficient detection performance.

The present study is organised as follows. In Chapter 2, first, we present the background to give an idea about the concepts related to our study. Then, we offer the relevant studies in the topics, including malware detection and language model. Chapter 3 describes our approach, datasets, the pre-trained model and binary classification model, GPT-2 (based on transformers architecture), train and test pipeline, parameters, and the setup of the environment used for training and testing. Chapter 4 reports the results, compares our pre-trained model and GPT-2's pre-trained models, and discusses the results. Finally, in Chapter 5, we present a conclusion, the limitations of the study, and the future work.

CHAPTER 2

BACKGROUND AND RELEVANT WORK

This chapter first presents deep learning (DL) methods used in natural language processing (NLP). Then, we describe the tasks performed using NLP. After that, we briefly summarize artificial neural networks (ANN) and its subsection. Lastly, we summarize approaches developed for detecting malware.

2.1 Deep Learning for Natural Language Processing

This section introduces deep learning (DL) methods used in natural language processing (NLP). NLP is a subcategory of computational linguistics. Linguistics focus on analyzing aspects of language, such as grammar, semantics, and phonetics, as well as the methods briefly summarize artificial neural networks (ANN) and its subsections for studying and modeling them (Contributors, 2019). However, computational Linguistics is concerned with understanding written and spoken language from a computational perspective. It formulates grammatical and semantic frameworks for characterizing languages. NLP aims at conducting the design and analysis of computational algorithms. The methods of NLP address human languages accessible to computers (Eisenstein, 2019). With the development of fast computers and the emergence of big data, novel findings have become available by processing large datasets of text by software. In the 1990s, statistical methods and statistical machine learning began to replace the classical top-down rule-based approaches to analyzing language. Employing their better results, speed, and strength, the statistical approach to studying natural language has dominated the field. Nowadays, statistical machine learning has evolved into deep learning neural networks to infer specific tasks and develop robust end-to-end systems. The basis of the NLP approach is that it identifies an *argmax* function that aims at finding the argument that gives the maximum value from a target function (Brownlee, 2020).

$$\nu = argmax\beta(x, y, \theta) \tag{2.1}$$

In (2.1), x is the input, an element of a set X, and y is the output, an element of a set Y(x). β is the scoring function also called the model. It calculates the argmax of β, meaning that the output ν that gets the best score given the input x. In other words, at this stage, the goal is to find the appropriate β value. θ is a vector of parameters for β, which can be conceived as the learning part of the process. ν is the predicted output (Brownlee, 2020). Thus, most NLP models aim to find the best score function with the best parameters for contexts. One of the key concepts of those models, especially in terms of artificial neural network (ANN) models known as classical neural networks rooted in the 50s (Contributors, 2018), is word vectors.Word vectors mean that every word in our vocabulary is mapped to a vector. The studies do our natural language analysis in the context of these word vectors. One-Hot Encoding and bag-of-words (BOW) are simple approaches to how to convert words into vectors. The One-Hot Encoding labels each unique word in a sentence or a document with an index. Each vector input is zero except one that is corresponding to its index set to 1. The index number is the number of different words in a sentence or a document. A more detailed approach compared to one-hot encoding is called BOW. This detailed approach means to count the occurrences and co-occurrences of all unique words in a context. Each part of a context, such as a sentence, is represented by a row in a matrix, where the columns are unique words of context. These approaches may be successful for a small number of unique words in the document. The word orders in sequences, such as sentences or context chunks, do not play a fundamental role, like in sentiment analysis, because these methods are independent of the word order in the sequences.

Moreover, word vectors usually generate high-dimensional with few non-zero values that are a problem for many machine learning models. To overcome these problems, models use word embeddings that are n-dimensional word vectors.

word embedding

The idea of embedding is to embed the word in n-dimensional feature space. The dimension of feature space, n, is depended on the task, such as a target word predicted in context. It also depends on vocabulary and computing capacity. More dimensions of feature space may improve the accuracy of the tasks; since the word embedding may capture more aspects of the word. Nonetheless, more dimensions also mean higher computing effort and time. It is for this reason that some algorithms are developed to determine the dimensions of feature space correctly. The two popular algorithms for calculating word embeddings are word2vec and GloVE.

In 2013, (Mikolov et al., 2013) introduced the two word2vec algorithms, which considerably influenced NLP models. To generate word embedding, word2vec uses a target word given its context, namely skipgram, or context words given a target, namely continuous bag-of-word (CBOW). In contrast, Glove uses local context windows, including global word co-occurrence counts, to generate vectors. Models learn syntactic or semantic similarities between those vectors and try to predict words or context.

In NLP, the feature space is called syntactic and semantic space that occurs similar words close to each other and more dissimilar words being far from each other. Machine learning algorithms based on ANN, particularly deep learning (DL) algorithms, want to generate the semantic space by themselves by evaluating the context of a word. This is more often done as an unsupervised procedure. These sets of vectors, in space, mapped to a large or massive corpus of unlabelled[1] documents. Then ML algorithms learn the relation between vectors.

In the following subsections, from NLP's perspective, we briefly summarize artificial neural networks (ANN) and its subsection, feed-forward neural networks and recur-

[1] Unlabelled: Unlabelled means to do not need humans interventions, such as reading documents.

rent neural network (RNN). Next, we explain transformers and generative pre-trained transformer 2 (GPT-2) (Radford et al., 2019) architectures primarily used for language model tasks based on the idea of deep learning currently. Afterward, we introduce a modern and efficient library, namely the Hugging Face transformers' library (Wolf et al., 2020).

2.1.1 Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs), as known neural networks, are developed to imitate biological neural networks. Just as the units of biological neural networks are neurons, the units of neural networks are nodes. However, nodes' functionalities are simpler than neurons'. It can be said that the nodes are small computing units that take input vectors and produce a single output value.

ANN is a sub-domain of machine learning, consisting of linked nodes. Each node in a neural network has a numerical value, named weight, mapped to an input used for prediction. The networks update weights of links between the nodes, depending on whether a prediction is an accurate one or not. In other words, the concept of learning in an ANN is represented by parameter updates. For updating the weight of a link, a commonly used technique is backpropagation. The network uses weights and inputs to predict the output and then calculate the actual and predicted output differences, also called loss. After then, the loss value spreads to all network node values by backpropagation. Consequently, the network achieves a learnable architecture (M.K., 2019).

Different neural network architectures are specialized to do different tasks. For example, the feed-forward neural network models aim to predict the next word based on a sequence of apriori words. Furthermore, recurrent neural network (RNN) is an extended version of the feed-forward neural networks that employ variable-length sequence input to achieve language modeling and speech recognition tasks. Nevertheless, convolutional neural networks (CNN) perform better on image data in pattern and image recognition tasks. Briefly, those networks' performance and success depend on the task on which they are performing.

2.1.2 Feed-forward Neural Networks in NLP

Feed-forward neural networks have two categories depending on the number of layers, single layer or multi-layer. A single layer feed-forward neural network has an input layer and output layer. Inputs are passed on to the output layer, and the output layer produces the results. Otherwise, multi-layer feed-forward neural networks have three layers: input, hidden, and output. The input layer takes the inputs, the hidden layer processes the inputs, and the output layer generates the result. The model may use more than one hidden layer. Each layer has multiple neurons, and their inputs are fully connected and processed only in the forward direction. E.g., (Bengio et al., 2003) used simple feed-forward neural networks. The model first learns the word embeddings in the input layer and hidden layer. In the second step, the output layer gives probability distribution over all words given a specific context.

On the other hand, since feed-forward neural networks use dense vector representations for words, namely word embedding, neural networks often learn those representations with multiple hidden layers. That architecture cause problem such as a calculation problem of loss value and fixed length of input and output. Besides, both feed-forward neural networks and word embedding techniques still do not take into account word order in context. For some NLP tasks, like sentiment analysis, this may not be a problem. Nonetheless, for other tasks like machine translation, the position of words in context can not be disregarded.

Recurrent neural networks (RNNs) solve this difficulty with cyclical connections, unlike the classical feed-forward neural networks. This means that multi-layer perceptrons (MLP), i.e., a feed-forward neural network composed of fully connected layers, can only map from input to output vectors. Conversely, RNNs provide the entire history of previous inputs to influence the network output.

2.1.3 Recurrent Neural Network (RNN) in NLP

Recurrent neural network (RNN) is a special artificial neural network that employs variable-length sequence inputs. In RNN, inputs are converted to vectors with special techniques, such as word embedding or one-hot encoding techniques. Those vectors process at nodes. Each node belongs to a network layer at a particular time situation. At time t, each node's hidden state $h^{(t)}$ gets its earlier hidden state $h^{(t-1)}$'s outputs and the new input vector v^t to calculate the current state outputs. Furthermore, the current state $h^{(t)}$'s output is transmitted to the next hidden state $h^{(t+1)}$ as an input (Goodfellow et al., 2016). The recurrently hidden states provide short-term memory for models.

The algorithm has different backpropagation from regular backpropagation in feed-forward neural networks because of the short-term memory architecture. Each node of the output is a function of the previous parts of the output, so backpropagation for the RNNs requires recursive gradient computations. It means that the gradient grows each step.

A gradient calculates how much the output function changes if the inputs are changed a little bit. In machine learning algorithms, a gradient calculates the change in all weight with regard to a loss function, i.e., the difference function between the expected state function and output state function. The expected and output functions are mapped weights and inputs at each time. Then recursively compute gradient causes some difficulties, such as very small gradient values known as exploding and very big gradient values known as vanishing gradients. They caused a complete loss of information about long-term dependencies. Long short-term memory (LSTM) networks were originated by (Hochreiter & Schmidhuber, 1997) to overcome problems of long-term dependencies.

2.1.4 Long Short-Term Memory (LSTM) in NLP

Long short-term memory (LSTM) solves the short memory problem,the gradient problems, by modifying the standard RNN cell structure with a read gate, a write gate, and a forget gate. Those gates provide information that deserves to be saved or removed. However, each memory unit (read, write and forget gates) affects every other unit's memory with learnable parameters, known as weight.

The more advanced version of LSTM is bidirectional long short-term memory networks (BiLSTM). Compared to LSTM, BiLSTM can train inputs bidirectional. BiLSTM is to stack two separate hidden layers. One hidden layer is responsible for the forward information flow, while another is for the backward information flow. All information also concatenates the final output. Thus, BiLSTM can access long-time dependencies in both input directions. Nevertheless, long sequential data is still a critical problem for LSTM. That means recursively compute gradient may still be a problem, i.e., exploding/vanishing gradient problem. In addition, while long sequential data is processed, memory constraints may limit batching across examples.

Conversely, the attention mechanism, particularly self-attention, proposes to process data in parallel, not sequential. It means that each element of sequential data can connect with other elements at the same time. Distant items can affect each other's output without recurrence. And also, due to transformers, items can attend multiple times the process.

2.1.5 Attention Mechanism

Attention was first proposed in neural machine translation (NMT) but in many models uses it, known as encoder-decoder architecture (Bahdanau et al., 2014). It is a problem for most other NLP tasks, such as text generation and text classification. From in DL perspective, attention can be interpreted as a vector of importance weights, known as an attention vector. We estimate using the attention vector how strongly a word is correlated with other words to predict or infer a word in sentences. We take the sum of their values weighted by the attention vector as the approach of the target.

In NMT, the attention mechanism allows each hidden state of the decoder in Figure 2.1 h^d to see a different and dynamic context, a function of all the encoder hidden states in Figure 2.1 h^e. Thus, the attention mechanism maps between source $\{x_1,x_2, ...,x_n\}$ and target $\{y_1,y_2,...,y_n\}$. While context vector controls and learns from an alignment between the source and target, it consumes three pieces of information: encoder hidden state, decoder hidden states, alignment between source and target. The hidden states are based on recurrent connections such as RNNs and LSTMs. Alignment function, aka alignment score function, computes the relation between encoder and decoder hidden states in the attention layer. In Figure 2.1, $\{c_1,c_2,\ldots,c_n\}$ are results of alignment function. There are three main attention mechanisms: Bahdanau attention (Bahdanau et al., 2014), Luong attention (Luong et al., 2015) provided in several variants in the original paper, and the transformers' self-attention (Vaswani et al., 2017). Their differences lie essentially in their architectures and computations. Bahdanau's and Luong's architectures are based on recurrent connections.

As we explain in previous subsections, passing information forward through an extended series of recurrent connections, such as RNNs and LSTMs, may cause difficulties in training. On the right side of Figure 2.2, the inherently sequential nature of recurrent networks limits the use of parallel computational resources. In contrast, on

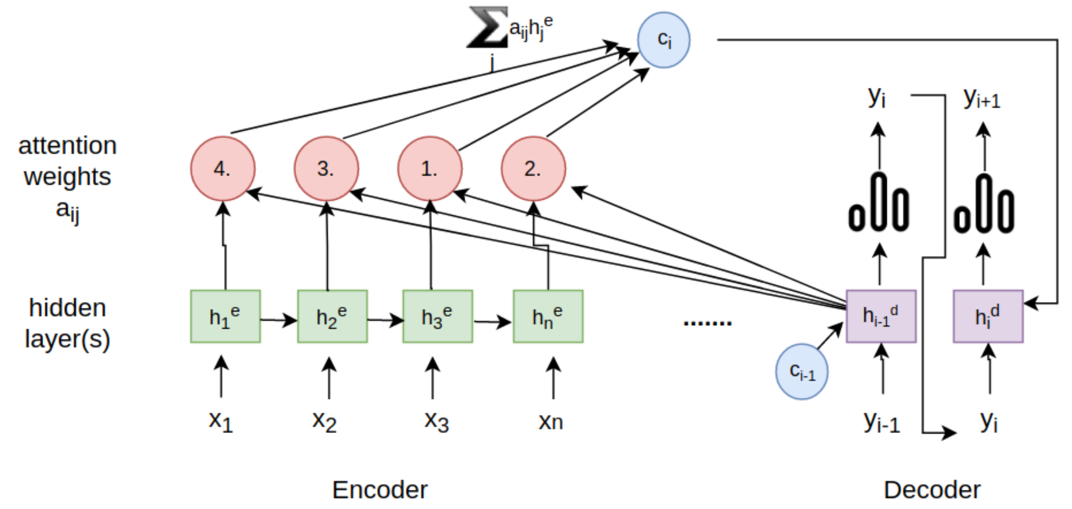


Figure 2.1: All hidden-layer based on RNN and h^{e}_{i} are encoder block and h^{d}_{i} are decoder block. i is an element of N that is the number of words in the context. (Lam, 2021)

the left side of Figure 2.2, each hidden state has dependencies on the previous words' hidden state. The embeddings of the current step are also generated one time step at a time.

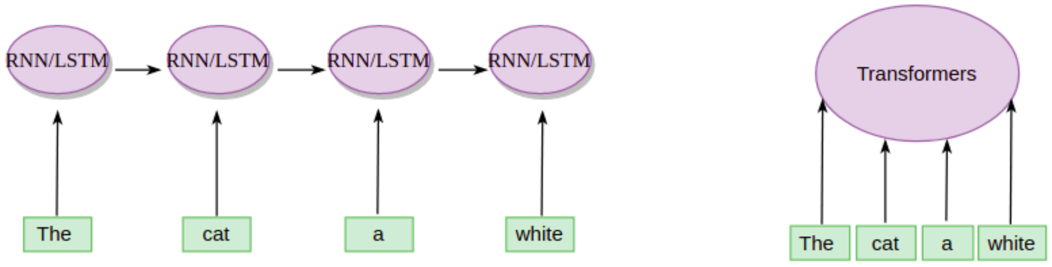


Figure 2.2: Compare RNN/LSTM and Transformers

On the other hand, self-attention network transformers eliminate recurrent connections and return to architecture recollections of fully connected networks. Therefore, the right side of Figure 2.2, the architecture processes arbitrarily large contexts in parallel. There is also no concept of the time step.

Self-Attention Mechanism

Self-attention enables a network to extract and handle information from arbitrarily large contexts directly. There is no need to pass extracted information through intermediate recurrent connections as in RNNs. Thus, no access to information about inputs beyond the current one. Nevertheless, while processing each item in the input, the model has access to all inputs, including the one under consideration. In short, we can efficiently parallel both inference, such as question answering and summarization, and training of language models since the computation performed for each item are independent of all the other computations.

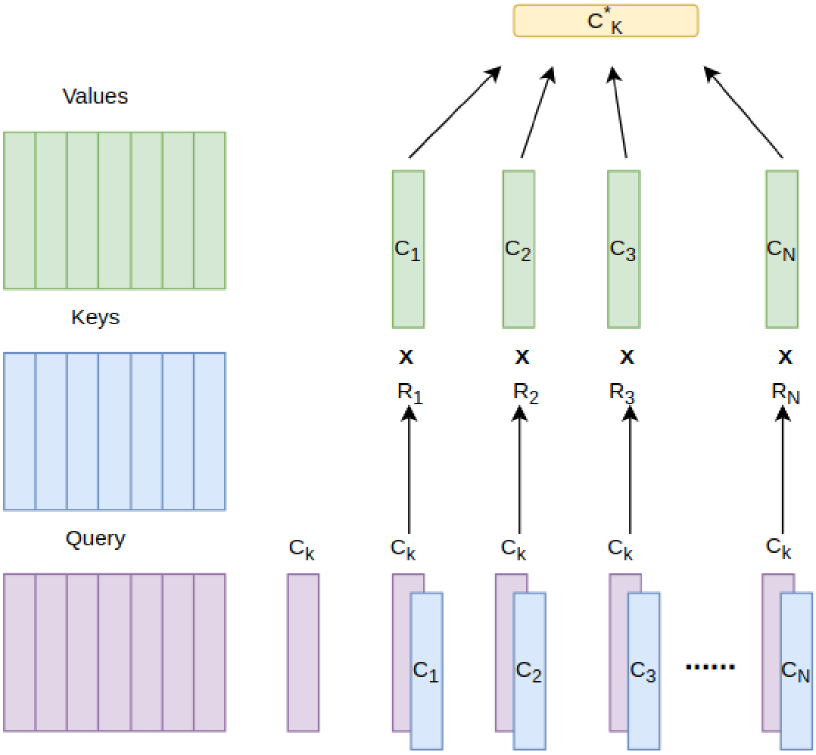


Figure 2.3: C^*_K represent query C_K in terms of weights R_i sum values, with weights define by match between query and keys. i and K is element of N that is the number of words in the context.

While self-attention extracts and uses information from context, it compares an item of interest, in Figure 2.3 C_K, to a collection of other items, in Figure 2.3 $\{C_1,C_2,\ldots,C_N\}$, in the current context. These comparisons reveal the relevance of C_K to $\{C_1,C_2,\ldots,C_N\}$ in the current context.

One of the easiest forms of comparison is between elements in a self-attention layer is a dot product. To do dot product, we use word embedding of words. The dot product results between C_K word embedding and $\{C_1,C_2,\ldots,C_N\}$ word embeddings are generated weight vector of target word embedding C_K, which is $[R_1, R_2,\ldots,R_N]$. Thus, self-attention attends to different positions C_K of the input sequence $\{C_1,C_2,\ldots,C_N\}$ to compute representations, in Figure 2.3 C^*_K , of input sequence using the weight vector of C_K. In short, new word embedding C^*_K is collection of weight vector and word embeddings $\{C_1,C_2,\ldots,C_N\}$, C^*_K=R_1C_1+ R_2C_2+...+R_NC_N.

The results of the dot product in $[R_1, R_2,\ldots,R_N]$, aka scores, can be negative or positive numbers-in other words, the scores range from -∞ to ∞. A negative score indicates that the word embeddings are far in space, so the words are less similar, whereas, as the outcome is positive, it shows that these words are similar.

The standard *SoftMax* function $\sigma : \mathbb{R}^N \rightarrow [0,1]^N$ is defined by the formula:

$$\sigma(R_\mathrm{i}) = \frac{e^{R_\mathrm{i}}}{\sum_{j=1}^{N} e^{R_\mathrm{j}}} \quad for\ i = 1, ..., N\ and\ R_\mathrm{i} \in [R_1, R_2, ..., R_\mathrm{N}] \qquad (2.2)$$

The architecture normalizes with a *SoftMax* function to use these scores effectively, range from -∞ to ∞. The *SoftMax* function, in (2.2) formula, takes the score vector [R_1,R_2,. . . ,R_N] as input. It normalizes inputs into a probability distribution consisting of score vector entry's probabilities proportional to the exponential of the input numbers. After applying the function, each component is in the interval [0,1], and the components add up to one.

This simple mechanism causes to need additional parameters for learning. In Figure 2.3, these new parameters are query, key, and value that have different roles. A self-attention function maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are word embeddings. The query is the current focus of attention, which is used to compute scores against all the other words embeddings. The key is a being compared to the current focus of attention. Finally, the value is used to compute the output for the current focus of attention. The query, key, and value are all generated from the same input embedding X.

Self-Attention Mechanisms With Scaled Dot-Product

In each of these steps, transformers introduce three sets of weights (W^Q, W^K, W^V) to capture the different roles that input embeddings play. These sets are matrices and learnable parameters of architecture. They are used to compute linear transformations of each input, with the resulting values being used in their respective roles in subsequent calculations.

For each query's score value defines;

q_i=$\mathrm{W}^\mathrm{Q}\mathrm{x}_\mathrm{i}$, k_i=$\mathrm{W}^\mathrm{K}\mathrm{x}_\mathrm{i}$, v_i=$\mathrm{W}^\mathrm{V}\mathrm{x}_\mathrm{i}$, X_i is a part of X input embeddings.

$$\alpha_\mathrm{ij} = score(x_\mathrm{i}, x_\mathrm{j}) = \frac{(q_\mathrm{i} \cdot k_\mathrm{j})}{\sqrt{d_\mathrm{k}}} \qquad (2.3)$$

As the score function defines in general, it gives similarity between x_i and x_j. x_i elements of queries set, x_j is elements of keys set. In (2.3) formula, the results of each q_i dot product to each k_j are divided by $\sqrt{d_\mathrm{k}}$.

The *SoftMax* may have an extremely small gradient for large input since the larger input components will correspond to larger probabilities. It is a vanishing gradient problem that causes to become hard for efficient learning. Accordingly, the square root of the depth $\sqrt{d_\mathrm{k}}$ scales the dot-product attention. Then, *SoftMax* is applied and the result is multiplied with v_i, to obtain the weight vector of q_i.

Given input embeddings of size d_m, the dimensionality of these matrices are $d_q \times d_m$, $d_k \times d_m$ and $d_v \times d_m$, respectively. In the original transformer work (Vaswani et al., 2017), d_m was 1024 and 128 for d_k , d_q and d_v.

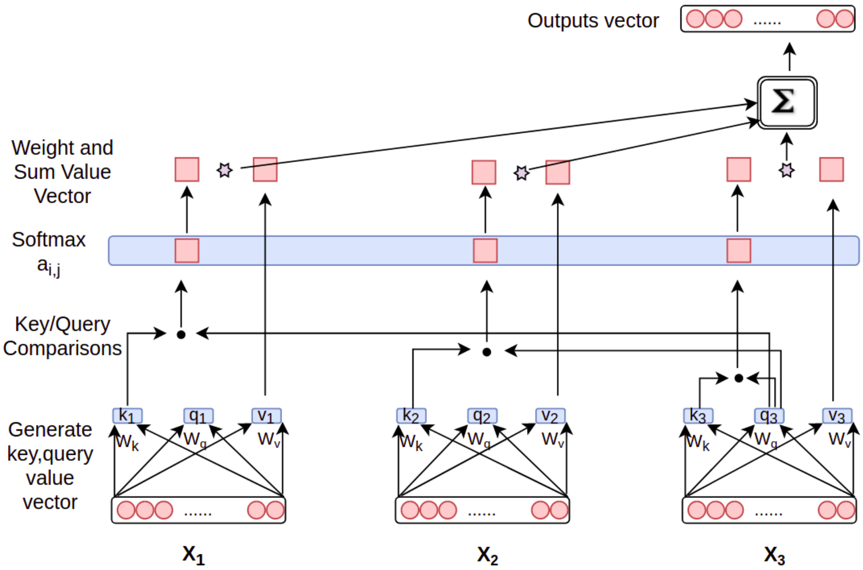


Figure 2.4: Self-attention (Lam, 2021)

In Figure 2.4, the output of the third element of a sequence is calculated using causal self-attention. The causal self-attention access only to the next vector v_i in the sequence of values set.

Self-Attention Mechanism - Parallelism

In (2.4) formula is general form of self-attention that computes on a set of queries concurrently, packed together into a matrix Q. Keys and Values are also packed together into matrices K, V, respectively.

$Q=W^{Q}X$, $K=W^{K}X$, $V=W^{V}X$

$$Self-Attention(Q,K,V) = SoftMax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.4}$$

By packing the input embeddings into a single matrix X and taking advantage of matrix multiplication can be parallelized an entire process. The calculation of the comparisons in QK^T results in the score for each query value to every key-value. Accordingly, the self-attention mechanism causes global dependencies between input and output with parallelization. In addition to the self-attention layer, we describe other features of transformer blocks.

2.1.6 Transformers

The transformer, based solely on self-attention mechanisms in NLP, is a new architecture that aims at solving the encoder-decoder network tasks by long-range dependencies. The encoder-decoder network of transformers avoids recurrence with fully connected layers,namely feed-forward layers. Transformers also map sequences of input embeddings to sequences of output embeddings of the same length.

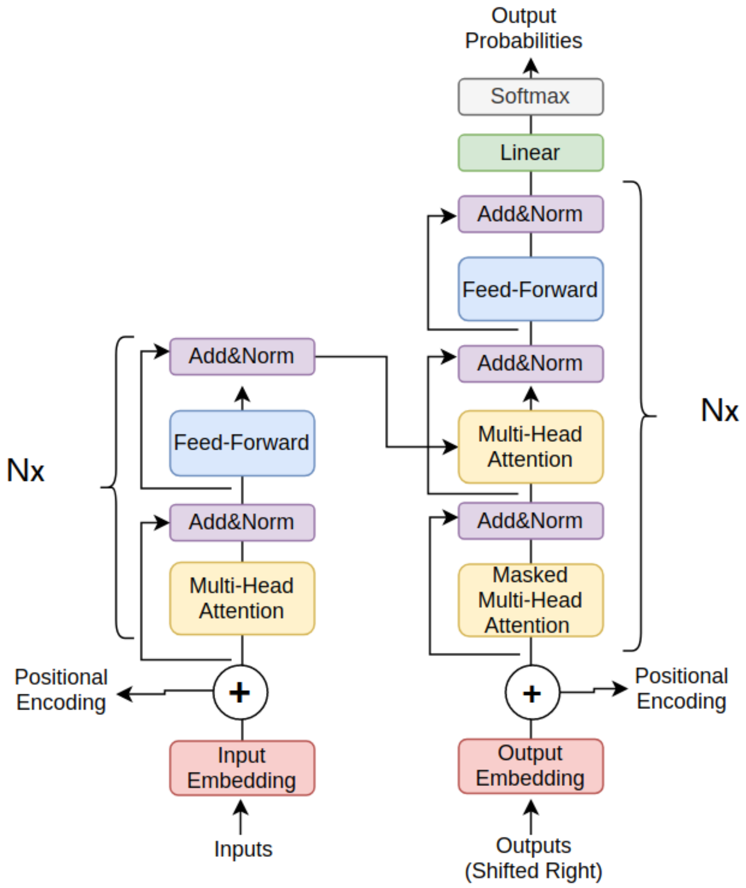


Figure 2.5: Architecture of Transformer (Vaswani et al., 2017)

In Figure 2.5, the left-hand side is the encoder parts of transformer. The right-hand side is the decoder part of transformer. In Figure 2.5, the encoder maps a sequence of input to a sequence of continuous representations. Given encoder output, the decoder then creates an output sequence for one element at a time. In the decoder part (auto-regressive part), each step consumes the previously created encoder output as additional input when producing the next. The transformer encode-decoder sides have a same structure except for one layer in the decoder, known as masked multi-head attention. This layer causes a real difference between the two blocks. First, we describe same layers and features in blocks, then the differences between blocks. The first same layer is the multi-head attention layer.

Multi-Head Attention

The architecture has performed single attention, also named the single-headed attention function, with keys, values, and queries. However, the different words in a sentence (input embedding) may relate to each other in many different ways simultaneously, i.e., words have distinct syntactic, semantic, and discourse relationships with other words and their arguments in a sentence. According to (Vaswani et al., 2017), it is difficult for a single transformer block to capture all of the different aspects of parallel relations among its inputs. Moreover, the model can not learn those parallel relationships. transformers approach this issue with multi-head self-attention layers, which reside in parallel layers at the same depth in a model, each with its own set of parameters. Hence, each head can learn different kinds of relationships that exist among inputs at the same level of abstraction with these distinct sets of parameters.

When implementing this notion, X input embedding divides into h, that is, the number of heads. Each head, i, in a self-attention layer is provided with its own set of the query, key, and value matrices: ${W_i}^Q$, ${W_i}^K$, ${W_i}^V$.These are used to project the inputs to the layer, x_i, distinctly for each head. The output of h heads consists of h embeddings of the same length.

$$head_i = Self - Attention({W_i}^Q X, {W_i}^K X, {W_i}^V X)$$

In (2.5) formula, each head output is concatenated and then reduced down to the original input dimension d_m by using another linear projection W^O,i.e., element of $h.d_v \times d_m$, to the original output dimension.

$$\begin{aligned} MultiHead(Q, K, V) &= Concat(head_1, ..., head_h)(W^O) \\ \text{where} \qquad head_i &= Self - Attention({W_i}^Q X, {W_i}^K X, {W_i}^V X) \end{aligned} \tag{2.5}$$

Owing to multi-head attention function, the model attends collectively to information from different representation subspaces at different positions.

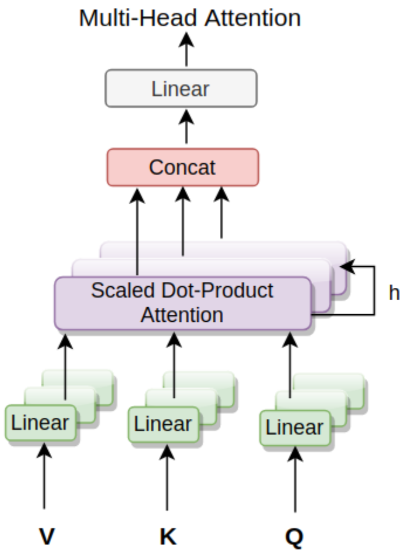


Figure 2.6: Performing h times attention function (Vaswani et al., 2017)

We may deduce in Figure 2.6 that multi-head attention includes several attention layers (h times) running in parallel. In this way, it captures h times long-term dependencies between elements in an input sequence.

On the other, unlike RNNs, transformer architecture layers, i.e.,multi-head attention and feed-forward, lose all information about elements' position in a sequence. Thus, the architecture can not use information about the relative or absolute positions of the elements of an input sequence. For solving this issue, the transformer architecture combines positional embeddings specific to each position in an input embedding.

Positional Encoding

Positional encoding allow the transformer structure to recognize that an input belongs to which part of the sequence. The original transformer (Vaswani et al., 2017) determines each word's position or the distance between different words in the sequence to generate positional embeddings using multiple sine and cosine functions, as in formula (2.6).

$$\begin{aligned} PE_{\text{pos,2i}} &= \sin(pos/10000^{\text{2i/d}}) \\ PE_{\text{pos,2i+1}} &= \cos(pos/10000^{\text{2i/d}}) \end{aligned} \tag{2.6}$$

d:Dimension size of word embedding, N: Number of word in the sequence, pos: position of the current word in the sequence in [0, N-1] and i: index of the dimensional index of word embedding in [0,d].

That is, each dimension of the positional encoding matches a sinusoid. It forms a geometric progression from 2π to 10000.2 π on the wavelengths. This function would allow the model to easily learn to attend by relative positions since $\text{PE}_{\text{pos+k}}$ can be represented as a linear function of PE_{pos}, for any fixed offset k (Vaswani et al., 2017).

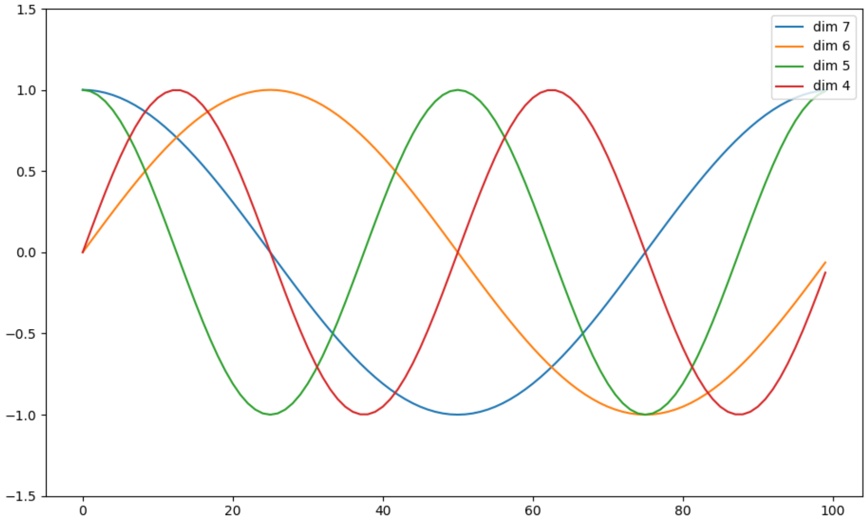


Figure 2.7: Sine and cosine graphs of dimensions (Shieber & Rush, 2018) N=100

Each position in each dimension,as in Figure 2.7, is mapped to sine and cosine functions that have a different frequency and offset of the wave. Since the sine and cosine curve only varies between a fixed range, it depends not on the input text length.

As a result, the network introduces new word embedding vectors that take into account the connection of a given word to its surrounding words. Through the positional embedding, while words are nearby positional, they will have similar positional embeddings and their inner products will be high. In contrast, while words are far apart, they will have different word positional embeddings and their inner products will be negative.

Nevertheless, multi-head attention and feed-forward layers also lose their original word embedding, the union of word embeddings and positional embeddings. The architecture solves this problem with the residual connection and is followed by normalization. These operations are called add and norm layers.

Residual Connection

In Figure 2.5 add and norm step, the add refers to a residual connection that adds the output of the residual connection to the output of the multi-head attention, i.e. original word embedding, and the norm refers to layer normalization. Residual connection, aka skip connection, provides knowledge prevention about original embedding (He et al., 2016). In addition, residual connection bypasses the gradient vanishing or exploding problem and proposes to solve the model optimization problem from the perspective of information transfer. It enables the integration and delivery of information by adding an identity mapping from the input embeddings to the output embeddings, which may ease the optimization (Liu et al., 2021). In deep learning (DL), optimization algorithms or methods used to change the attributes of the network such as learning rate [2] and weights to reduce the losses.

Layer Normalization

Transformer places the layer normalization between residual blocks (Vaswani et al., 2017), as in Figure 2.8.

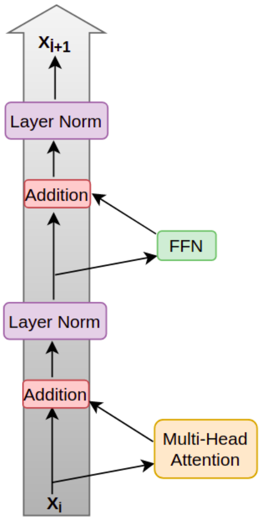


Figure 2.8: Residual blocks (Xiong et al., 2020)

[2] en.wikipedia.org : https://en.wikipedia.org/wiki/Learning_rate (retrieved on 4 Sep 2021)

This architecture has achieved state-of-the-art performance in language modeling (Al-Rfou et al., 2018). Unsupervised pre-trained models, such as Generative Pre-trained Transformer 2 (GPT-2), based on the layer normalization between the residual blocks also show impressive performance in many downstream tasks (Radford et al., 2019).

Layer normalization (LN) normalizes the results of intermediate layers, known as multi-head attention sub-layer or feed-forward neural network sub-layer, with residual connection outputs. It means that LN progresses by normalizing and rescaling the intermediate layer (hidden layer) result to handle the collective optimization of multiple correlated features. Thus, it enables faster training and can help optimize the non-linear transformation. The model also obtains better generalization accuracy (Xiong et al., 2020). In (Vaswani et al., 2017), LayerNorm function is defined as

$$LayerNorm(x + Sublayer(x))$$

x is outputs of residual connection, and sub-layer is multi-head attention or feed-forward neural network.

Let x = (x_1, x_2, . . . , x_N) be the word embeddings of an input of size N to normalization layers. LayerNorm re-centers and re-scales input x as

$$LayerNorm(x) = \gamma \cdot \frac{v - \mu}{\sigma} + \beta$$

,

in which μ, σ are the mean and standard deviation of the elements in x ,i.e.,

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_{\mathrm{i}} \quad \text{and} \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{\mathrm{i}} - \mu) \tag{2.1}$$

scale γ and bias (shift) vector β are parameters.

Therefore, Layernorm uses all of the summed inputs to the activation result of neurons (features) in a layer to compute the mean and variance for every sub-layers (Al-Rfou et al., 2018).

LN and skip connection are used techniques to facilitate the optimization of deep neural networks (Liu et al., 2021). In (Vaswani et al., 2017), the network uses LN and skip connection, and also Adam optimizer (Kingma & Ba, 2014) with a learning rate warm-up stage for controlling the gradient since the self-attention mechanism creates unbalanced gradients. In the warm-up stage, the optimization schedule starts with an extremely small learning rate. The schedule then gradually increases the learning rate to a pre-defined maximum value in a pre-defined number of iterations (Xiong et al., 2020). The number of iterations is 4000 in the original paper. Consequently, utilizing the warm-up stage, the adam optimizer fixes the unbalanced gradients problem by having different learning rates for each parameter.

In Figure 2.8, in addition to the multi-head attention sub-layer, residual connection and layer normalization, each transformer layer contains a fully connected network, which is applied to each position separately and identically.

Position-wise Feed-forward Neural (FFN) Sub-layer

This sub-layer involves two-layer linear transformations with a ReLU activation function between them. Given a sequence of vectors h_1,h_2 ..., h_n, the computation of a position-wise FFN sub-layer is defined as:

$$FFN(h_i) = ReLU(h_i W^1 + b^1)W^2 + b^2$$

W^1, W^2 weight matrices of each layer and b^1,b^2 are bias matrices of each layer.

$$ReLU(x) = max(0, x) \qquad (2.7)$$

Furthermore, *ReLU* is an activation function. In formula (2.7), the function takes an unbounded real input and returns a non-linear transformation of that input between 0 and 1 differentiable to the input.

The last common point found from both the encoder and decoder part is the dropout parameter for regularization. The original paper applies dropout (Srivastava et al., 2014) to the output of each sub-layer before it is added to the sub-layer input and normalized. In addition, the paper author applies dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, they use a rate of P_{drop} = 0.1.

In Figure 2.5, the difference between the encoder and decoder stacks are masked self-attention sub-layer, an extra one linear layer, and *SoftMax* activation at the end of the decoder stack. The linear layer is a feed-forward network that gives the relational score of the words, and *SoftMax* gives probabilities of words. The main difference between encoder and decoder blocks is the style of self-attention in the masked self-attention layer. This layer also causes a difference between encoder-based and decoder-based transformers variations.

Differences between Encoder-Based and Decoder-Based Transformer

First, to tell the difference between encoder-based and decoder-based transformers, we should explain the difference between language models (LM) and masked language models (MLM). LMs like Generative Pre-trained Transformer 2 (GPT-2), i.e., models with a unidirectional architecture, predict the next word on given the previous words in the input sentence. Utilizing decoder-based blocks, LMs handle input sentences from the start of the input to the end of the input (right-hand-side of Figure 2.9). Instead of predicting the next word, MLMs attempt to predict a masked word selected randomly from the input (left-hand-side of Figure 2.9, such as "Bidirectional Encoder Representations from Transformers (BERT)" (Horan, 2021). MLMs are bidirectional since they access words before and after a current word to predict a word. Therefore, each encoder block of MLMs consists of a bi-directional self-attention layer. In contrast, each decoder block consists of a unidirectional self-attention layer (masked-self attention). The decoder-based transformer is a stack of decoder blocks followed by the LM head, and the encoder-based transformer is a stack of encoder blocks followed by the MLM head (Platen, 2020).

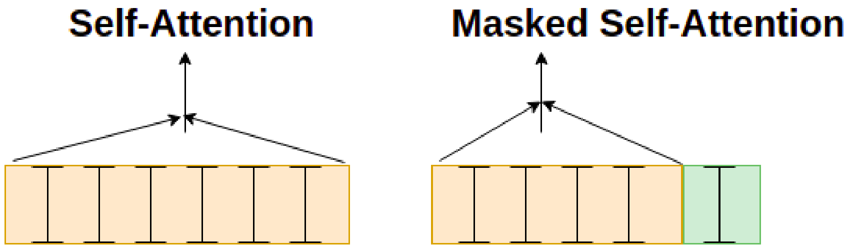


Figure 2.9: The left-hand side is MLM architectures attention mechanism. The right-hand side is LM architectures attention mechanisms (Becker et al., 2020)

The success of the self-attention mechanism in determining the relationship between input and output leads to the emergence of new architectures for NLP tasks such as machine translation, text classification, and summarization. Generating models for these tasks uses the encoder-based transformer and the decoder-based transformer, which are based on different styles of self-attention mechanisms. Furthermore, these transformer-based architectures have different tokenizer styles, models, and architectures' configuration files. Since we use GPT-2 decoder-based transformer for our classification task, in the following subsection, we primarily describe the infrastructure of GPT-2. However, at first, we give brief information about transfer learning since it allows us to leverage knowledge acquired from related data to improves performance on our classification model based on GPT-2.

2.1.7 Transfer Learning

The availability of large amounts of data and increased computation resources make it challenging to train models. In NLP, one of the common solutions for this challenge is Transfer learning that mainly consists of two steps: pre-training and adoption. As in Figure 2.10, the model is first pre-trained with the source task, and second, the model is adapted, i.e., is adopted model, to the target task, such as text classification and question answering.

The pre-trained language model holds structural and semantic relations between each word and every other word in context. The word embedding layer captures morphological information, and the lower layers capture local syntax in pre-trained models. On the other hand, the upper layers capture longer-range semantics (Peters et al., 2018). Pre-trained language models have also been shown to learn logic rules (Krishna et al., 2018). These rules give us weight parameters of a language model, which hold long-range semantics of a language. Therefore, to improve the adopted model's performance and reduce the training time, the pre-trained models' weights are employed as a starting point for the target task rather than building a model from scratch

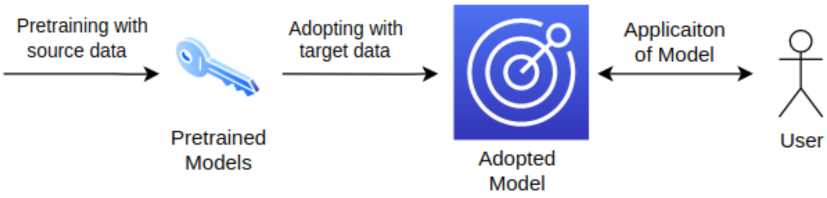


Figure 2.10: Transfer Learning (Becker et al., 2020)

to solve similar problems. As in Figure 2.10, in transformers-based models, the pre-trained language model's gained knowledge, i.e., weights parameters, transfers to the adopted model during training used for NLP tasks. This phase is called fine-tuning, which updates the pre-trained representations. The user can use these adopted models for the designated task.

We describe the architecture of Generative Pre-trained Transformer 2 (GPT-2) since our language model pre-training (pre-trained language model), custom language model pre-training (custom pre-trained model), binary classifier fine-tuning (adopted model) are based on GPT-2. These three type of models have same tokenizer style and same tokenizer model.

2.1.8 Tokenization for Generative Pre-trained Transformer 2 (GPT-2)

Natural language Processing (NLP) begins with words, corpora, and tokenization. A collection of texts used to train an NLP model is called corpora, and a simple definition of words is sequences of letters. Tokenization splits words, sentences, and corpora into smaller parts according to specific rules that use whitespace characters or other means. An output of a tokenizer is tokens. Words can be part of a token or sub-token of a token. The number of distinct tokens is called types.

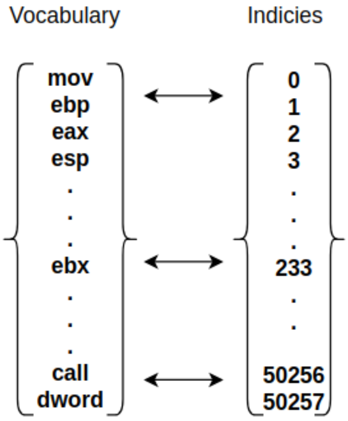


Figure 2.11: Vocabulary and their indices

To obtain word embeddings, in Figure 2.11 tokenization maps tokens to numerical values. It also collects vocabulary having uniques tokens of texts and obtains indices of vocabulary. Hence, it generates word embeddings using indices. For instance, suppose that sentence is [mov eax,ebx] then word embedding of the sentences is [0,2,233].

Tokens depend on an actual tokenizer since all transformer-based models have their tokenizer styles. GPT-2's actual tokenizer is based on byte-pair encoding (BPE), which was introduced in (Sennrich et al., 2015). BPE is a kind of pre-tokenizer depending on a set of rules to split inputs. In the pre-tokenizer phase, GPT-2 splits on white-spaces while mapping all the bytes to a set of visible characters. Since it maps on bytes, the tokenizer uses 256 characters as an initial alphabet. Therefore, GPT-2 can represent every word, including rare and unknown words, as sequences of subword units with 256 tokens.

GPT-2's BPE model runs by starting with characters while merging the most frequently seen together, thus creating new tokens. Then, it works iteratively to produce new tokens out of the most frequent pairs it sees in a corpus. For instance, it iteratively counts all symbol pairs and replaces each occurrence of the most frequent pair ('X,' 'Y') with a new symbol 'XY'. It does so until the vocabulary has attained the desired vocabulary size that is a hyperparameter of the tokenizer model. In this way, the BPE model can build words it has never seen using multiple subword tokens, thus requiring smaller vocabularies, with fewer chances of having "unk" (unknown) tokens. For this reason, GPT-2's BPE model is called the subword tokenization model.

2.1.9 Generative Pre-trained Transformer 2 (GPT-2)

This part is based on (Radford et al., 2019). GPT-2 is an open-source artificial intelligence (AI). It is created by OpenAI, in February 2019, which is an AI research and deployment company for artificial intelligence. In open source library have language model that Large-scale unsupervised language model GPT-2 trained on a large dataset containing 8-million web pages including 40 GB of internet text. There are also models with relatively smaller scales compared to this model,as shown in Table 2.1. Moreover, GPT-2 is an extended version of GPT (Openai et al., 2018) with more than ten times the parameters and trained on more than ten times the amount of data. It succeeds in state-of-the-art performance on many benchmark language model tasks. It can perform machine translation, question answering, language generation, and summarization tasks (Radford et al., 2019).

GPT-2 is a decoder-based transformer. Due to the decoder-based structure, GPT-2 is a unidirectional language model based on,shown in Figure 2.12, Autoregressive language (AR) model (Yang et al., 2019). Given a text sequence X=(x_1,...,x_T), AR model factorizes the log-likelihood into a forward sum;

$$\log P(x) = \sum_{t=1}^{T} \log P(x_{\mathrm{t}} | X_{<\mathrm{t}}) \qquad (2.8)$$

Table 2.1: Architecture hyperparameters for the 4 models size based on GPT-2 (M is the abbreviation for million)

Parameters	Layers	Dimension of Input
117 M	12	768
345 M	24	1024
762 M	36	1280
1542 M	48	1600

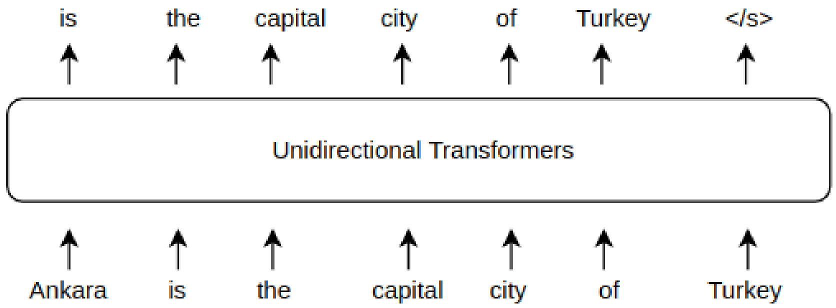


Figure 2.12: Auto-regressive language model (Becker et al., 2020)

In this way, (2.8) formula estimates the probability distribution of a text sequence with an auto-regressive pattern.

The language model has an unsupervised pre-training followed by supervised fine-tuning for specific NLP tasks, and its training objective is formulated as P(output|input). On the other hand, instead of the fine-tuning model with specific tasks, GPT-2 aims at learning multiple tasks using the same unsupervised model through many self-attention blocks. While the unsupervised model sets without any parameter or architecture modification to understand the multiple tasks, this is known as a zero-shot setting. Utilizing a zero-shot setting, the learning (training) objective is modified to P(output|input, task), so the model produces different outputs for the same input for various tasks. GPT-2's perform down-stream tasks in a zero-shot setting. As a result, GPT-2 is not trained on any of the data specific to any of the tasks, such as translation, classification, question-answering, and is only evaluated the tasks as a final test (Radford et al., 2019).

To perform better under the zero-shot setting, training GPT-2 needs enormous data.Hence, researchers create a new dataset webtext, which contains over 8 million documents for a total of 40GB of webpages text. During the representation of webtext (tokenization process), GPT-2 models use Byte-Pair Encoding. Hence, the unknown character

(<UNK>) infrequently occurs in the webtext tokenization in GPT-2.

Moreover, GPT-2's model extends the OpenAI GPT model with some improvements. Larger model GPT-2 has 48 layers and uses 1600 dimensional vectors for word embedding. The layer normalization from the transformer is moved to the input of each sub-block, and an additional layer normalization is added at the end of the final self-attention block. A larger vocabulary of 50,257 tokens is used. At initialization, the weight of residual layers is scaled by $(1/\sqrt{N})$, where N is the number of residual layers. Instead of ReLU, GELU is used as an activation function to provide a higher probabilistic and avoid vanishing gradients problems (Hendrycks & Gimpel, 2020).

2.1.10 Hugging Face

Hugging Face has an extendable framework and an open-source NLP library using by production. The open-source NLP library name is transformers which opens their advances to the wider machine learning community. The library is dedicated to supporting transformer-based architectures and facilitating the distribution of pre-trained models. For this reason, the transformers generates python-based application programming interfaces (APIs) for many well-known transformer architectures, such as BERT, RoBERTa , or GPT-2. These transformer models involve different shapes, sizes, and architectures. The models also accept input data in different ways. Every model in the library is fully defined by three building blocks shown in the diagram in Figure 2.13 a tokenizer such as byte-level BPE based on BPE encoding, a transformer such as GPT-2, a head such as language modeling, sequence classification, and question answering (Wolf et al., 2020).

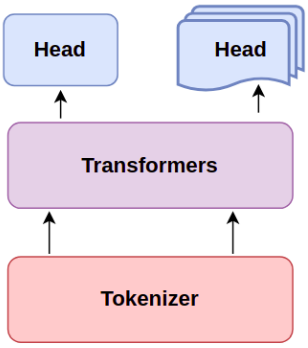


Figure 2.13: Main blocks of every model in the transformers library (Wolf et al., 2020)

Everything is shared, and everyone can contribute to the Hugging Face framework, so it is easy to read, extend, and deploy. Hugging Face organizations also support the distribution and usage of various pre-trained models in a centralized model hub.

In the following section, we summarize malware analysis and methods, and the approaches developed for detecting malware.

2.2 Malware Analysis

2.2.1 What is Malware

Malware is a code that executes malicious actions. It takes the form of an executable, a script, or it is embedded in other software. It attempts to violate the system's or device's security policies by executing it on a system or computing device. In this way, attackers use these violations to steal sensitive information, spy on the infected system, or take control of the system.

Malware is categorized, such as trojans, viruses, worms, and rootkits, based on their functionalities and attack vectors used to detect malware. Nevertheless, the malware authors and attackers try to modify or morph their malware to evade malware detection. Their standard techniques for evading detection are Polymorphism, Metamorphism [3], Encryption, and Packing [4] that are obfuscation techniques. Whether these techniques are used, two steps are processed to detect malware: feature extraction and classification/clustering.

The performance of malware detection methods critically depends on these processes. Firstly, various features are extracted statically, dynamically, or a hybrid to capture the characteristics of samples. Different features such as opcodes (description of assembly-code-level operations), byte sequences (description of byte-level contents), system/APIs calls (analyses of execution traces, disassembly code, and description of APIs' executed actions), memory accesses (during malware executions, analyses of memory), file system (frequency of created-deleted-modified files) and network traffic (like analyses of incoming, outgoing packets and visited addresses) are being applied to malware analysis (Akhtar et al., 2015). In the second step, machine learning (ML) techniques, in particular deep learning (DL) techniques, are used to automatically categorize samples into different classes/groups based on the analysis of feature representations (Ye et al., 2017).

2.2.2 Malware Analysis Methods

Malware analysis dissects malware to understand how it works and to find solutions to eliminate it. There are two fundamental approaches for analysis: static analysis and dynamic analysis, which have intelligent techniques to extract features representing files.

Static Analysis

Static analysis is collecting information about the malicious application without running it. It has two categories as basic and advanced. Basic static analysis is achieved by extracting all the possible static information inside the file, such as the hash, libraries, strings, imported functions, and resources. Therefore, analysts capture a basic understanding of the functionalities and the behaviors of suspicious files before their

[3] Polymorphism and Metamorphism: the code changes itself every time it runs.

[4] Packing: packed and encrypted malware.

execution. Nevertheless, advanced static analysis, aka code analysis, inspects the binary file to study each component, still without executing it. The advanced static analysis also provides more information about the characteristics of malware than basic analysis.

For dissecting the binary file, one method is to reverse engineer the binary file code using disassemblers such as IDA, objdump. Through disassemblers, sub-sections, i.e., different regions and thus types of data, are obtained from the PE file. The PE format specifies various sections types, such as .text that contains executable code, .data for initialized variables, and .idata for the import table, for storing information needed for the program to execute. Malware detection techniques use code patterns of these disassemblers' outputs, such as assembly instructions, opcodes sequence, opcodes frequency, byte n-grams, and n-gram of the opcodes, obtained from malicious files and benign files. For example, Bilar (Bilar, 2007) determined that the difference between malware files and benign files was statistically significant in opcode frequency distributions. Furthermore, they used rare opcodes as a predictor for malware detection.

However, the static analysis fails to discover malware intents owing to different code obfuscation techniques used by malware authors. Hence, dynamic analysis is suitable for analysing code obfuscation to inspect different functionalities and behaviors.

Dynamic Analysis

In malware detection, analysts get execution traces from suspicious files during execution, extract the features from traces and interpret these features to detect malware. For example, in (Acarturk et al., 2021),researchers constructed a dataset in instruction format from run trace outputs acquired from dynamic analysis of PE files. They fed into models with instructions and reached an accuracy beyond 99% with Basic Block as a Sequence Model (BSM) for malware detection.

However, in dynamic analysis, each malicious file should be run in a controlled environment such as sandboxes or virtual machines to protect against possible infections. That causes to need a specific time to monitor the behavior. The monitoring also requires a high amount of scanning time. Moreover, the secure environment may be different from a real runtime environment, and the malware may behave differently in the two environments. It is for this reason that the trace of the malware behavior may be incorrect. In addition, some malware actions are triggered under certain conditions such as system date and time or some particular input by the user, which the secure virtual environment may not detect (Egele et al., 2012). Consequently, both static analysis and dynamic analysis use for different problems, so their outputs are examined in different malware detection methods.

2.2.2.1 Malware Detection Methods

Malware detection approaches are divided into two major categories that consist of behavior-based and signature-based methods. Each category has advantages and disadvantages. We briefly explain them.

Signature-Based Methods

Signature-based detection methods rely on identifying known malware. While a signature-based detection provider identifies an object as malicious, It adds its signature to a database of known malware. This database may contain millions of signatures identifying malicious objects. Identifying malicious objects method has been the main technique used by malware products. Furthermore, it continues the base approach used by the latest firewalls, email, and network gateways.

Signature-based malware detection technology has several strengths, such as its integrity, i.e., following all possible execution ways of a given malware, simple to run, speedy, and widely available. First of all, it provides good protection from the many millions of older but still active threats. Nonetheless, the signature-based malware detection technology has disadvantages, such as the inability to recognize newly produced malware, the unmanageable growth of signature databases, and the troublesome signature generation and distribution processes (Souri & Hosseini, 2018).

Behavior-Based Methods

Behavior-based detection concentrates on malware activities as a system is infected. These malicious actions may be file activities, registry activities, API calls, control flow graphs (CFGs), and system calls. Therefore, behavior-based detection improves the detection of new malware variants. Its methods use two ways to classify activities. The first way includes extracting behavioral characteristics statically from the malware code. For example, (Preda et al., 2008) presented a technique based on the structural analysis of binary code that allowed one to identify structural similarities between different polymorphic worm mutations. The approach was based on the analysis of a worm's control flow graph. The second way is to run malware in a sandbox environment and dynamically monitor its behavior. For instance, system calls establish malware behavior monitored dynamically and then used for malware detection. In (Lin et al., 2015), authors defined malware behavior vectors and calculated the cosine similarity[5] to classify the malware. In this way, their architecture distinguished the known-type malware with an accuracy of 85.8%. Nonetheless, behavior-based detection takes a long time, and resources may be intensive as malware or benign runs in virtual machines or sandboxes. Furthermore, false positives are often a concern due to the misclassification of benign software since benign and malware may exhibit similar behaviors.

2.2.2.2 Machine Learning and Deep Learning Based Methods

The rise in new malware variants requires automating malware detection since it can no longer be detected with human resources. Therefore, automated detection methods are needed against malicious software with little or no human intervention. For this reason, the focus of academic studies in malware detection has evolved from traditional methods(signature-based and behavior-based methods) to machine learning classification methods since the last decade and, in the previous few years, from machine learning detection methods to deep learning neural networks. In machine learn-

[5] Cosine similarity: Cosine similarity is a metric measuring similarity between two non-zero vectors of an inner product space

ing detection methods, the detection process is usually two-step: feature extraction and classification(supervised learning algorithms)/clustering(unsupervised learning algorithms). The performance of such malware detection methods critically depends on the extracted features and the categorization techniques. In the feature extraction phase, various features such as opcode frequency and sequence, binary strings, API calls, CFG, and program behaviors are extracted statically and/or dynamically to capture the characteristics of the file samples (Ucci et al., 2019).

In our study, we discuss the detection of Windows Portable Executable (PE) files. We decompile executables by obtaining assembly instructions to feed our models. Programming languages, including assembly instructions, have clear grammar and syntax. Thus we may treat them as natural language, process by NLP models. We ultimately may use the experience in modeling the natural languages to model the assembly languages. We find some pieces of malware to be extensively similar to each other if they are from the same family. In addition to malware detection, in malware samples, our approaches can also help us to find consistent patterns and locate malicious payloads. Numerous recent studies focus on assembly code for feature extraction. In the second step, intelligent techniques such as classification or clustering are used to automatically categorize the file samples into different classes/groups based on the analysis of feature representations.

Training samples, including malware and benign files, are provided to the system in the classification process. The extracted features representing sample files underlying characteristics are converted to vectors in the training set. Both the feature vectors and the class label of each sample (i.e., malicious or benign) are used as inputs for a classification algorithm (e.g., artificial neural network (ANN), support vector machines (SVM), logistic regression (LR), naive bayes classifier (NB), decision tree (DT), boosted tree (BT), and random forest (RT)) (Gibert et al., 2020). By analyzing the training set, the classification algorithm builds a classification model. Then, new suspicious file samples representative vectors are presented to the classification model generated from the training set. Next, the model will classify the new suspicious file samples based on the extracted feature vectors using the same feature extraction techniques as in the training phase. Moskovitch (Moskovitch et al., 2008) presented a method for classifying malware based on text categorization techniques. The paper's authors extracted all n-grams [6] from the training data that are binary code of files, with n ranging from 3 to 6. Then, they selected the top 5500 features according to their TF inverse document frequency (TF-IDF) for feature selection[7] phase. Afterward, using the resulting features as input, they trained various algorithms such as an SVM, NB, ANN, and DT. Their results indicated that greater than 95% accuracy could be achieved by using their training set. (Shabtai et al., 2012) proposed a framework to detect malware based on opcode n-gram features with n ranging from 1 to 6. They performed a wide set of experiments to: identify the best term representation, whether it was the term frequency (TF) or term frequency-inverse document Frequency, determine the n-gram size, find the optimal K top n-grams and feature selection method and evaluate the performance of SVM, LR, RF, ANN, DT, NB, and their boosted versions, BDT and BNB, for n;6 and classifier; RT, best accuracy 95,6% with TF.

[6] n-gram: n-gram is a contiguous sequence of n items from a given sequence of text

[7] feature selection: feature selection is the process of selecting a subset of informative and relevant features from a larger collection of features for use in model construction.

Furthermore, (Santos et al., 2010) studied the frequency of opcode sequences. They studied the frequency of opcode sequences. While they analyzed the relations among the opcodes to detect variants of known malware families, they employed statistical information, such as the frequency of appearance of opcode sequences.

In many cases, very few labeled training samples exist for malware detection. Thus, researchers have proposed clustering to automatically group malware samples that demonstrate similar behaviors into different groups. Clustering is the task of grouping a set of objects such that objects in a cluster are more similar, such as using certain distances, to each other than those in other clusters. In the malware detection studies centering on machine learning clustering, the k-means clustering algorithm is the most used machine learning clustering algorithm (Ucci et al., 2019). In (Pai et al., 2016), the researchers applied clustering techniques to the malware classification problem. They used k-means and expectation maximization algorithms (i.e., unsupervised learning techniques) to compute clusters with the underlying scores based on Hidden Markov Models. Their results showed that classification accuracy in excess of 90% was easily achievable. Another work (Nataraj et al., 2011) extracted GIST features from the grayscale representation of binary content of malware. Their model was based on the K-nearest neighbor algorithm (K-NN) with the Euclidean distance. Then, they used this model to classify new executables under one family or another. In (Santos et al., 2011), researchers used both supervised learning and unsupervised learning. The method was based on analyzing the appearance of frequency of opcode sequences to create a semi-supervised [8] machine learning classifier using a set of labeled and unlabeled data. A few other successful works, in (Zhang et al., 2019) features were extracted from n-gram opcode sequences. Moreover, five different machine learning classification algorithms were used to detect and classify ransomware families. The authors were the first to propose an approach based on static analysis to classifying ransomware. While they used real datasets, their approach achieved the best accuracy of 91.43%. In addition, the average F1-measure of the WannaCry ransomware family was up to 99%, and the accuracy of binary classification was up to 99.3%. Those studies reached high accuracy values. In (Yewale & Singh, 2016), the best accuracy was 97% with the RF algorithm. Malware unpacked and disassembled using *UPX*[9] Unpacker and IDA pro, respectively. Their features were Opcodes Frequency.

Academicians have ensured active interactions with machine learning algorithms to learn from and make predictions by features extracted from malicious or benign software through machine learning. The usage of computational statistics and mathematical optimization in algorithms have provided reliable and fast results by learning. In contrast, the traditional malware detection methods based on machine learning require human control over feature extraction and feature engineering, leading to time-consuming machine learning workflow processes. Human intervention also disrupts the automated detection process (Gibert et al., 2020). At this point, deep learning techniques, in other words, neural networks, come in popular in malware analysis because they learn information from data by themselves with reducing human intervention or without feature extraction by humans. Hence, the neural network, which

[8] semi-supervised: semi-supervised learning combines both labeled and unlabeled data for feeding models to gain knowledge

[9] UPX: UPX is an executable packer for several executable formats.

becomes automatic, shows better results in some malware detection cases. In malware detection studies using deep learning techniques have various approaches. These existing methods have two different aspects. First is the recurrent neural networks (RNN) and their derivations, and convolutional neural networks (CNN); second is the transformers relying on attention mechanisms. The effectiveness of these methods depends on the input features extracted from the dataset. CNN is the first neural network that regards distances in the input or output sequences. It employs feature extraction by converting malware into images, as CNNs are mainly used for image recognition. For instance, Krcal et al. (Krčál et al., 2018) treated 20 million unpacked half megabyte Portable Executables (PEs) as a sequence of bytes and applied a convolutional network for malware detection. The network had four convolutional layers and four fully connected layers. Instead of a global max-pooling layer, they used a global mean pooling layer after the convolutional layers. Moreover, their best afford was 97.1% accuracy. In (Khan et al., 2018), Khan et al. studied GoogleNet and employed images obtained from opcodes of binary files for five different ResNet models. They improved images to distinguish between benign and malicious opcode images with Histogram standardization augmentation and resolution techniques for easy detection. The GoogleNet accuracy ratio was 74.5%, and among ResNet models, the top accuracy ratio was 88.36%. In (Kumar et al., 2018), The authors classified malware opcode images employing a model based on CNN. Their model achieved 98% accuracy in classifying binary files.

The second neural network is RNN. The data collected from malware are put into a sequential format, as in text classification tasks, to use on RNNs since it shows better performance on sequential data. The authors of (Jha et al., 2020) focused on step size as an essential factor with input size using RNN. They tested the model with three different feature vectors. Their results showed that RNN with word2vec feature vector achieved the highest area under the curve (AUC) value and a good variance among the three feature vectors. (Lu, 2019) used opcodes and operands data, which can be used as features, mapped with different word embedding techniques to word vectors. The researchers used word embedding results to feed into their models for malware detection; the work relied on long short-term memory (LSTM), complex gated RNN architectures with a long-term dependency of features. For malware detection, that model succeeded in an average AUC of 0.99, while for classification, the model achieved an average AUC of almost 0.99. Other study that fed their models as in our study, instruction2vec (Lee et al., 2019) work used both opcode and operand information to classify malware. They used a nine-dimensional feature vector to resemble registers and addresses. They split assembly instructions and encoded each token as unique index numbers. In their setup, an opcode takes one token, a memory operand took up to four tokens, including base register, index register, scale, and displacement. This approach represented information about opcode and operands.

On the other hand, transformer-based models, defined with attention mechanism, for malware detection are increasing in the literature since data collected from malware is processed in parallel rather than in a sequential format. The work in (M. Li et al., 2021) used benign and malicious assembly code obtained from the static content of an executable. Interpretable MAlware Detector (I-MAD) based on transformers model combined network components called the Galaxy Transformer network that identified assembly code at sequences of assembly instructions, sets of sequences of assembly instructions, and sequences of bytes. It also consolidated their proposed interpretable feed-forward neural network to provide solutions for its detection results by measuring the impact of each feature on the prediction. They found 97.7% accuracy. In another work (Rahali & Akhloufi, 2021), researchers classified different malware categories by focusing on the source code of Android applications through static analysis. They used the transformers-based model BERT for the classification model. In conclusion, they obtained 97.61% accuracy with BERT, yet they achieved 94.05 with LSTM.

2.3 Summary

In this section, we presented background information about malware analysis, neural networks, natural neural network (NLP), and the relevant works on malware detection. Malware analysis and NLP are the two main parts of this study. Since we focus on malware detection by applying NLP techniques, it is crucial to understand certain parts of malware analysis and NLP. In addition, we explained neural networks in detail, particularly how transformer-based model and generative pre-trained transformer 2 (GPT-2) works since they are the specific methods that we used to model assembly code. We shared the relevant works related to malware detection, including earlier statistical approaches and machine learning-based and deep learning-based methods. In studies use popularly nowadays. In the scope of this study, we investigated the assembly language to apply NLP techniques. Specifically, we collected dissemble Windows executable files and obtained assembly code as our dataset. Next, we performed the transfer learning (pre-trained) model for fine-tuning on the obtained dataset by applying GPT-2 architecture and named as our custom pre-trained model. In the second part, we created a binary classification model architecture. Then, we investigated how the assembly code format affects the success of the binary classification models based on GPT-2. In order to succeed in the binary classification models, we used custom pre-trained model to fine-tune the binary classification models. In the next section, we present the details of our methodology.

CHAPTER 3

METHODOLOGY

In this chapter, first, we present our approach to malware detection. Then, we introduce datasets and methodology of data collection, besides adapting data format for our models. Next, we follow with the introduction of the language model for assembly codes. Finally, we introduce the binary classification model fine-tuning with the knowledge of the language model.

3.1 Approach

Malware detection methods usually begin with the feature extraction processes, which perform through static, dynamic, or hybrid analysis. While approaches based on dynamic analysis and hybrid analysis work by executing Portable Executable (PE) files to examine their behavior, the static analysis looks at the content of executable files without requiring their execution. The static approach of PE files may provide a large set of significant information such as sections, imports, symbols, and used compiler strings. In traditional studies, the methods extract signatures stored in signature databases from this information via human intervention. Security systems compare these signatures with the signature of an executable file that the system has newly encountered and determine whether the executable is malicious or not. The signature-based malware detection is straightforward and fast, yet it may be ineffective against sophisticated malware or overlook relations. Moreover, the database of signatures grows too quickly to keep up with the growth rate of new malware.

The machine learning (ML) algorithms, in particular deep learning (DL) algorithms, are deployed to eliminate the drawbacks of traditional malware detection. DL is the end-to-end learning approach, which refers to training a possibly complex learning system represented by a single model, a deep neural network (DNN). The network represents the complete target system, automating feature extraction nearly with no pre-processing. In our study, we extract assembly code using an open-source disassembler *objdump* to create an opcode sequence as output. We employ the output as raw data to build a language model assisted with word embedding, just like processing natural language. Utilizing this language model, we aim to adopt polarity detection methods to identify the intention of an executable file using the labels as malicious and benign. Therefore, we plan to detect whether it is malicious or benign with our proposed language model. In the next section, we introduce the datasets used for developing the models.

3.2 The Datasets

At the beginning of work, dataset consisted of benign and malicious executables in Portable Executable (PE)[1] format. Then, we obtained new databases from these executables collecting from various sources.

3.2.1 Data Collection

We generated native Win32 PE files from Windows operating systems[2] and Commando VM v-2.0[3]. We chose Commando VM over the rest of the versions of Windows OS, because it contains executables compiled using different compilers, such as Cygwin5[4] and MinGW6[5]. Malicious executables were downloaded from the VirusShare website. Malicious samples included various type of malware such as virus, worm and trojan.

In our study, we used assembly code obtained from disassembled PE files. The PE is derived from COFF (Common Object File Format)[6], and it contains headers and sections. Headers, such as PE header and optional header, are the rules that represent windows loader on how the section should be mapped and loaded into memory. Sections consist of the data or content. Section content is the actual code that is required. It contains resources, data and code and other executable information. Different sections hold different kind of data, e.g. code section contains executable code while .rdata contains read-only data like constants and string literals. We used the executable code section, namely a .text section, to construct our dataset.

We focused on assembly instructions obtained statically from the collected executables. In the first step, we disassembled each benign and malicious file to get assembly instructions contained in the code section. Next, the output were saved in plain text files. The first dataset comprised of these plain text files merged without labeling and later used for pre-trained model. The second dataset included rows of assembly instructions, which were labeled as benign or malicious and later used for the binary classification model training. The overall processing pipeline is presented in Figure 3.1.

3.2.2 Data Formating

In the data processing pipeline, we created two different datasets for the present study. The first dataset consisted of unlabeled 10 M (million) assembly instructions, 5 M (million) samples from malicious executables and 5 M samples from benign executables as shown in Table 3.1.

[1] PE Format: https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail (retrieved on 20 Sep 2020)

[2] (Microsoft Windows 8.1 Pro (OS Build 9600), Microsoft Windows 10 Pro 19.09 (OS Build 18363.418)

[3] Commando VM: https://github.com/fireeye/commando-vm(retrieved on 2 Apr 2019)

[4] Cygwin: https://www.cygwin.com (retrieved on 20 Sep 2021)

[5] MinGW: http://mingw-w64.org (retrieved on 20 Sep 2021

[6] COFF: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (retrieved on 13 Sep 2020)

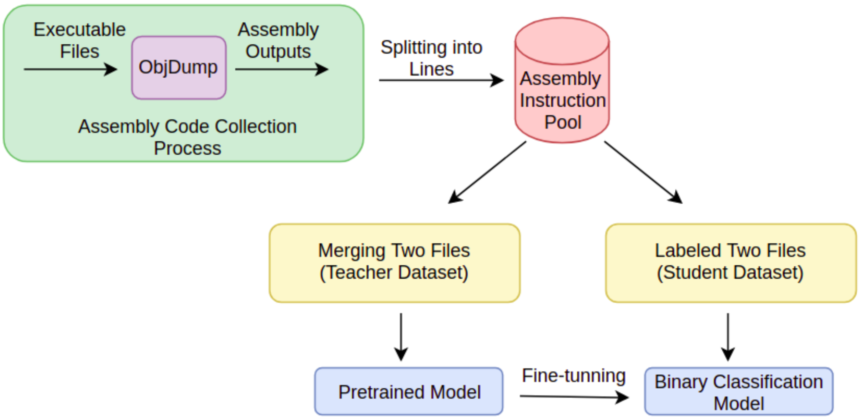
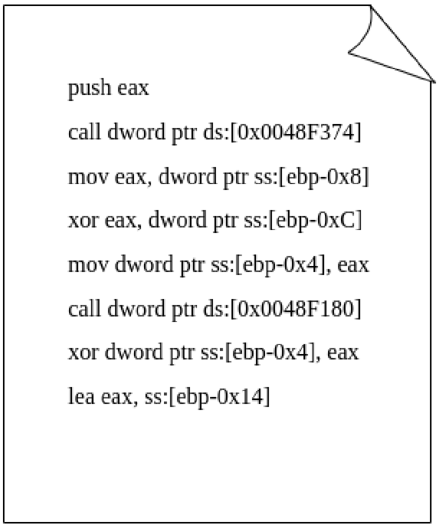

Executable Files
ObjDump
Assembly Outputs
Assembly Code Collection Process
Splitting into Lines
Assembly Instruction Pool
Merging Two Files (Teacher Dataset)
Labeled Two Files (Student Dataset)
Pretrained Model
Fine-tunning
Binary Classification Model

Figure 3.1: The data processing pipeline

Table 3.1: Characteristics of datasets (M is the abbreviation for million)

Number of instructions for	Malicious	Benign	Total
model creation (teacher)	5 M	5M	10 M
detection (student)	8 M	8 M	16 M

We used the unlabeled first dataset for creating language model. Figure 3.2 is the sample rows of merged text.



```
push eax
call dword ptr ds:[0x0048F374]
mov eax, dword ptr ss:[ebp-0x8]
xor eax, dword ptr ss:[ebp-0xC]
mov dword ptr ss:[ebp-0x4], eax
call dword ptr ds:[0x0048F180]
xor dword ptr ss:[ebp-0x4], eax
lea eax, ss:[ebp-0x14]
```

Figure 3.2: Sample data for model training

The second dataset, which we used for binary classification model, consisted of two text files as benign.txt and malicious.txt. The benign.txt consisted of 8 M((million)) benign assembly instructions, and the malicious.txt contained 8 M malicious assembly instructions. In training phase, we revised the assembly instructions of these files, as shown in Table 3.2, by labeling them as malicious or benign. Moreover, we fed models with assembly instructions, e.g. Figure 3.2, composed of opcodes and operands as conducted in a previous study (Lee et al., 2019). The researchers used a nine-dimensional feature vector to mimic registers and addresses. They split assembly instructions and encoded each token as unique index numbers. In their setup, an opcode took one token whereas a memory operand took up to four tokens, including base register, index register, scale and displacement. This approach represented information about opcode and operands. However, the reduction of the features such as encoding all addresses caused more similarity between feature vectors than it should have (X. Li et al., 2021). Hence, we decided to use all opcodes and operands as they were.

Table 3.2: Sample from the binary classification model dataset (M is the abbreviation for million)

Id	Content	Label
0	jecxz 0x402046	benign
1	dec eax	benign
2	add BYTE PTR ds:0x26b800,al	benign
3	add BYTE PTR [eax+0x1c],al	benign
....		
16M-4	sub esp,0xdc	malicious
16M-3	mov DWORD PTR [ebp-0x18],eax	malicious
16M-2	mov eax,DWORD PTR [ebp-0x24]	malicious
16M-1	sub ecx,0x907c8453	malicious

3.3 The Model

This section introduces technical specifications relevant to the models in our study, including the training and testing environment setup, used libraries and modules, and details of the pre-trained model training pipeline, binary classification model based on Generative Pre-trained Transformer 2 (GPT-2) training and testing pipeline. Before giving the details about the technical parts of study, we should briefly explain why we prefer to use GPT-2 based on Transformers over other neural network architectures such as recurrent neural networks (RNN), long short-term memory (LSTM). The neural network architectures, especially LSTM, are preferred for natural language processing (NLP) tasks since they perform better than other neural network architectures like RNN and Convolution Neural Network (CNN). RNN has a short memory to remember the previous situations, which causes performance problems while processing long sequences. There are also vanishing and exploding gradient issues in the standard RNN architecture. LSTM, a special RNN architecture, solves the gradient problems and improves standard RNN by modifying the cell structure. However, long sequential data is still a critical problem for LSTM.

In contrast, the transformer processes data more efficiently on short texts. It defines a new state-of-the-art with an attention mechanism that provides global dependencies between input and output. By using short text instead of long sequential data, the transformer-based model deals with gradient problems. A text classification with neural network GPT-2, based on transformers architecture, approaches the malware detection problem from an NLP perspective. There are other effective transformer-based architectures such as RoBERTa, BERT, XLnet [7]. We preferred GPT-2 for binary classification and pre-trained models because of BPE tokenizer and decoder-based transformers architecture.

3.3.1 The Environment Setup

We used Collaboratory by Google[8] , a Jupyter notebook-based runtime environment, to run code entirely on the cloud. There were 1 GPU(s) available. We used the GPU: Tesla V100-SXM2-16 GB, and training time was limited to 24 hours.

Table 3.3: Required Python libraries

Library	Version
Pytorch	1.7.0
Transformers	3.5.1
Matplotlib	3.1.3
Sklearn	0.22.2

3.3.2 Imported Libraries and Modules

In this section, we will give a short explanation of the python libraries, the Hugging Face libraries and their modules that are used in the scope of our study.

import tokenizers

A tokenizer is generally responsible for preparing the inputs for a model. The Hugging Face library contains tokenizers for all the models. Most tokenizers have a full python implementation, or a fast implementation relies on the rust library. In modern language model, the rust-tokenizer offers high-performance tokenizers (Wolf et al., 2020).

[7] XLnet, Generalized Autoregressive Pre-training for Language Understanding.

[8] Google Collab Pro: allows to write and execute Python in browser.

from tokenizers import ByteLevelBPETokenizer

The *ByteLevelBPETokenizer* model is a full python implementation model and based on byte-pair encoding algorithms. Output of the model are such as ids, type_ids, tokens, attention_mask, special_tokens_mask. They are used for various purposes. E.g., some models' goal is to classify pairs of sentences using ids, attention_mask.

from transformers import GPT2TokenizerFast

GPT2TokenizerFast model is a fast-implementation model backed by Hugging Face's tokenizers library, whose output is only 'ids'. The fast implementation allows significant speed-up for doing batched tokenization.

from transformers import GPT2Config

GPT2Config is the configuration class and creates the configuration file of a GPT2Model according to the specified arguments, defining the GPT-2 model architecture.

from transformers import GPT2LMHeadModel

GPT2LMHeadModel class is utilized a left-to-right language model (auto-regressive model).

from transformers import LineByLineTextDataset

LineByLineTextDataset, which splits data into chunks, returns as each line is interpreted as a document. It has block_size hyperparameters. When determining block_size length, we must be careful not to overstep a line length.

from transformers import DataCollatorForLanguageModeling

DataCollatorForLanguageModeling is used for language model. If inputs are not all of the same lengths, data collators have dynamically padded to the maximum length. mlm hyperparameter handles masked language model (MLM) when the mlm flag option is True, and the auto-regressive language model (LM) no mlm flag option is False.

from transformers import Trainer, TrainingArguments

The Trainer class provides an API for feature-complete training and eval loop, defined in Pytorch[9] and optimized for Transformers. The TrainingArguments is the subset of the arguments which depends on the training loop. We define the trainer hyperparameters according to database and training purposes.

Import torch

The module is a python package. It provides tensor computation with strong GPU acceleration and a deep neural network based on a reverse-mode automatic differentiation used to compute gradients and happens to be used by backpropagation.

from sklearn.metrics import accuracy_score, classification_report

[9] Pytorch : https://pytorch.org (retrieved on: 21.01.2019)

The sklearn.metrics module implements classification reports and it calculates accuracy score to measure classification performance.

from Transformers Import set_seed, adamw, get_linear_schedule_with_warmup, GPT2ForSequenceClassification

set_seed hyperparameter uses for reproducibility. Adamw[10] is an optimizer used for training. *Get_linear_schedule_with_warmup* create the learning rate scheduler. GPT2ForSequenceClassification is a transformer-based model for a sequence classification using the last token to do the classification.

3.3.3 Pre-trained Model

This study builds a pipeline to take the dataset, processes the data for the model and trains it with the neural network. To construct the pipeline, we implemented it on Python 3.7.11 by using Pytorch and Hugging Face open-source libraries. In the following subsection, we give details of the pipeline[11]. As discussed in the datasets section, we merge the assembly instructions extracted from PE files named as merged.txt. We use merged.txt to feed language model based on GPT-2, as in Figure 3.3.

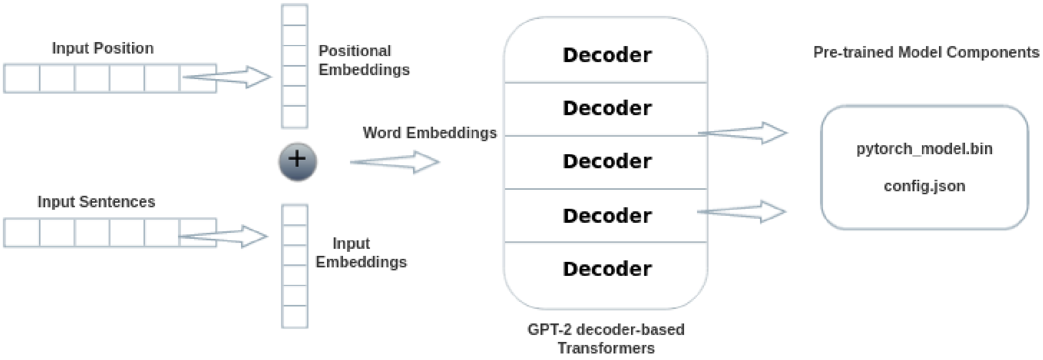


Figure 3.3: Pre-trained model based on GPT-2

Language Training Pipeline

We use *run_language_modeling.py*[12] scripts that create Hugging Face's transformers language model to form our custom pre-trained model. Firstly, we needed to define a merging rule of language (merges.txt) and a language dictionary (vocab.json). For creating those files, we used *ByteLevelBPETokenizer*, a byte-level encoding tokenizer.

In the first stage, to create those files, *ByteLevelBPETokenizer*'s parameters are customized: vocabulary size (dictionary size) as 50257, minimum frequency of the tokens as two, sentences (assembly instructions) and special tokens. Special tokens are

[10] Adamw: https://arxiv.org/abs/1711.05101

[11] The script will be shared upon request: https://github.com/nazeninsahin/Malware-Detection-GPT. (retrieved on: 20.08.2021)

[12] Script: https://github.com/huggingface/transformers/blob/master/examples/legacy/run_language_modeling.py (retrieved on: 25.06.2021)

composed of beginning of the sentence (<*bos*>) token, end of the sentence (<*eos*>) token,the unknown token (<*unk*>),the padding token (<*pad*>) and the mask token (<*mask*>). Models learn information about short-texts via special tokens. For example, the model stops generating more words if the architecture runs into a special end-of-sentence (<*eos*>) token. Alternatively, the architecture uses (<*unk*>) token for representing an out-of-vocabulary token[13]. After feeding with unlabeled assembly instructions corpora (merged.txt) to the *ByteLevelBPETokenizer*, the tokenizer builds merges.txt and vocab.json.

Tokenization

ByteLevelBPETokenizer firstly splits the training data into words that is pre-tokenization phase of ByteLevelBPETokenizer, and in GPT-2 architecture, pre-tokenization is defined by space tokenization. After pre-tokenization, a set of unique words are produced, and the frequency of each unique word contained in the training data is determined, such as (..,(DWORD, 300),(eax, 572),(ebx,519),..). Next, *ByteLevelBPETokenizer* creates a base vocabulary including all symbols, such as [..., D, W, O, R, D,...], and in the set of unique words, such as DWORD. Furthermore, it learns merging rules to build a new symbol, such as OR, from two symbols of the base vocabulary, such as (O, R). It does until the vocabulary reaches the desired vocabulary size (50257).

For example, the *ByteLevelBPETokenizer* starts with two symbol characters (O R) to reach the DWORD opcode word since it calculates that the letters O and R are frequently together in the language, using the frequencies of the characters or subwords. Then, to obtain DWORD tokens' merge rules, a new token OR are added to the base vocabulary(vocab.json), and to keep the merge rule, (O R) pair are added to merges.txt. Next, the architecture determines that W and OR are frequently together; it adds WOR to the vocab.json and (W OR) pairs to the merges.txt file. It does so until DWORD opcode is achieved.

Parts of vocab.json;(.,"OR":269,"WOR":270,"WORD":271,"DWORD":272,"int":273,.)

Parts of merges.txt; (..,a x, m o, mo v, O R, W OR, WOR D, D WORD, i n, in t ..)

To illustrate, we tokenize ("sub eax,DWORD PTR [eax]") sentence with *ByteLevelBPETokenizer*. The output is as follows:

tokens=('sub', 'Ġeax', ',', 'DWORD', 'ĠPTR', 'Ġ[', 'eax', ']')[14] and input ids= [352, 277, 16, 272, 264, 265, 280, 65]

Consequently, the byte level tokenizer creates vocab.json consisting of all symbols obtained from the set of unique words and merges.txt consisting of a list of the most frequent tokens ranked by frequency. Thus, we can follow *run_language_modeling.py* scripts for generating the pre-trained model. Here, we defined a tokenizer of our pre-trained model with vocab.json and merges.txt.

Utilized Gpt-2 Models

For the tokenization of the pre-trained model, we used *GPT2TokenizerFast* because

[13] Special tokens: https://huggingface.co/transformers/main_classes/tokenizer.html(retrieved on: 25.06.2021)

[14] The model knows spaces between tokens using Ġ

of its performance. The Hugging Face library has two types of tokenizers that are *GPT2TokenizerFast* and *GPT2Tokenizer*. Those tokenizers are also used for determining maximum sequence lengths. We determined the maximum sequence length of assembly instructions as 45 tokens using *GPT2TokenizerFast*. In addition, we set vocab.json and merges.txt to the pre-trained model's tokenizer.

Next, we defined a configuration file that had hyperparameters of the pre-trained model of GPT-2. They were vocabulary size 50257, number of hidden layers 12, dimensionality of the embeddings and hidden states 768, number of attention heads for each attention layer 12, activation function gelu instead of relu. The dropout probability for all fully connected layers, the dropout ratio for the embeddings, and the dropout ratio for the attention was 0.1. In the layer normalization layers, the epsilon was 1e-05. All hyperparameters of model was default values of GPT-2, which was 117 M hyperparameters. (M is the abbreviation for million). While we fine-tuned the binary classification model, we used this same configuration file.

After defining the configuration file with the default configuration of GPT-2, we initialized the model with *GPT2LMHeadModel*[15]. Finally, as in Figure 3.3, we obtained an auto-regressive model based on a left-to-right language model for the trainer. In addition, we set the mask language model (mlm) as false (unmasked model) because our aim was to predict the next word on given the previous words in the input sentence. Hence, the pytorch_model.bin file including model's weights and config.json file including configuration hyperparameters were created to be used for improving fine-tune binary classification model.

Parameters for Language Model

We determined that the maximum length of sentences was 128 tokens as sentence's maximum length was 45. 128 was the smallest value of a maximum length in GPT-2 LM architecture. Moreover, we selected that the *per_device_train_batch_size parameter* as 32, which depended on the batch size per GPU for training. On Google collab pro GPU, we trained with 32 and 64. In addition, 32 gave less loss value than 64.

3.3.4 Binary Classification Model

We constructed GPT-2 based-model for the classification of benign and malicious sentences, namely assembly instructions, using Hugging Face transformers. In addition, to improve the classification phase, we used our custom pre-trained model's knowledge.

Binary Classification Model Train and Test Pipeline

Following a similar approach we adopted in pre-trained model, for creating binary classification model, we build a pipeline to take the dataset, prepare the data for modeling, and train and test the neural network. To construct the pipeline, we implemented it on Python 3.7.11 by using Pytorch and Hugging Face open-source libraries. In the following subsection, we give the details of the pipeline.

[15] GPT2LMHeadModel: https://huggingface.co/transformers/model_doc/gpt2.html (retrieved on: 25.06.2021)

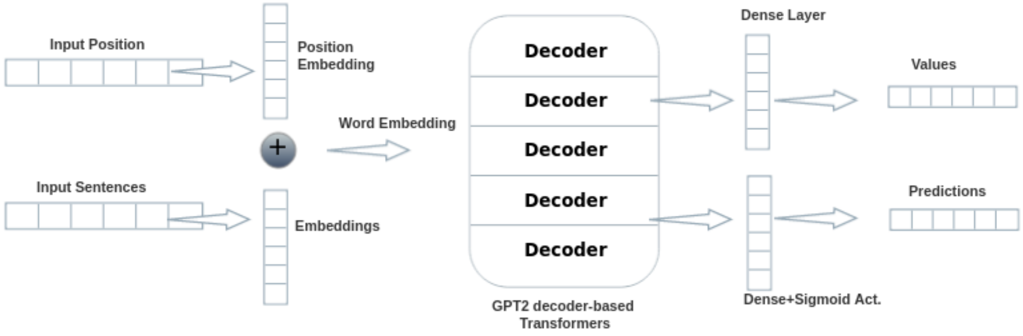


Figure 3.4: Binary classification model based on GPT-2

As discussed in the datasets section, we extracted the assembly instructions from benign and malicious PE files named as benign.txt and malicious.txt. We used benign.txt and malicious.txt to feed binary classification model based on GPT-2 as in Figure 3.4. We labeled the sentences in benign.txt and malicious.txt as malicious or benign at the next stage, i.e., we labeled malicious sentence lines with 0 and benign sentence lines with 1. Thus, we returned texts with their associated labels. We illustrated a row from the database as follows:

dataset.texts[0]=[sub eax,DWORD PTR [eax]] dataset.labels[0]=0

To construct a binary classification model, we loaded three essential parts of the GPT-2 transformer: model configuration file, model tokenizer and an actual model. We specialized these parts for malware detection.

We set the label number field as two with the purpose of classifying files as benign or malicious. Our custom pre-trained model's config.json file is defined as the configuration file of the binary classification model. We also defined vocab.json and merges.txt files as the binary classification model's tokenizer. After creating the tokenizer, we introduced the special tokens of GPT-2 to the tokenizer. Since the last token of the input sequence contained all the information required in the prediction in GPT-2, we set the tokenizer to pad the left side of sentences, and its pad token was |<*endoftext*>|. Hence, we used that information for the classification task. Then, in the tokenizer step, since training times depended on the length of sentences, we first tokenized each sentence of the benign and malicious and added |<*endoftext*>|. While the length of the sentences was shorter than the maximum sequence length of 45, the tokenizer padded sequences with eos and encoded them. Thus, as the model did not have to truncate the encoded sequence during the training, it sped up.

One of the dataset row; **label**: 0, **text**: [sub eax,DWORD PTR [eax]]

One of the encoded row; Its Label Tensor -> 0 Its Attention Mask Tensor -> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]

Its input ids Tensor -> [50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 50255, 337, 278, 16, 272, 264, 265, 284, 65]

Tensor is just a generic n-dimensional array to be used for numeric computation. Attention mask argument dictates to the model which tokens should be included to and which should not. Input ids are numerical representations of token sequence that is used as input by the model (Wolf et al., 2020).

For example, due to the Hugging Face library, GPT-2 model obtained weight parameters from layer seven by these input ids as follows;

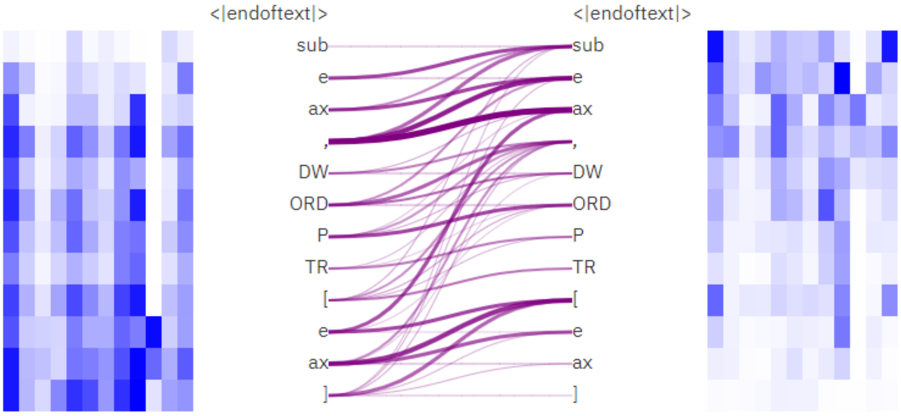


Figure 3.5: In layer 7, all relation of sentence's tokens with all heads [16]

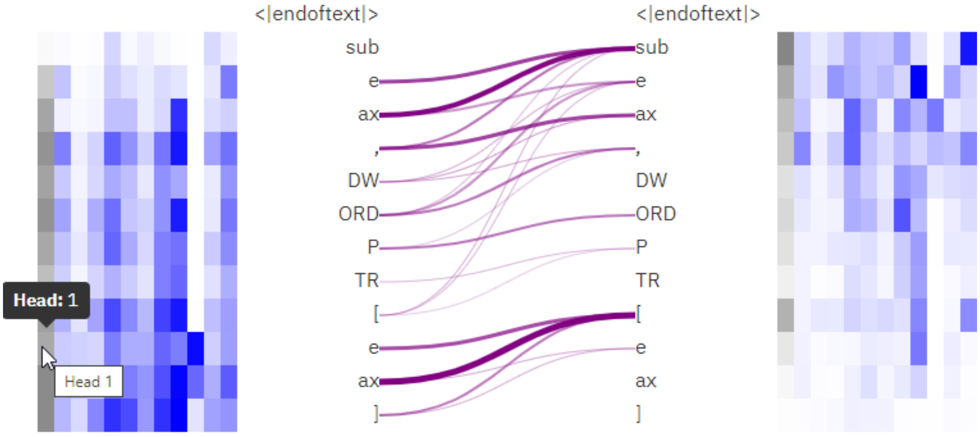
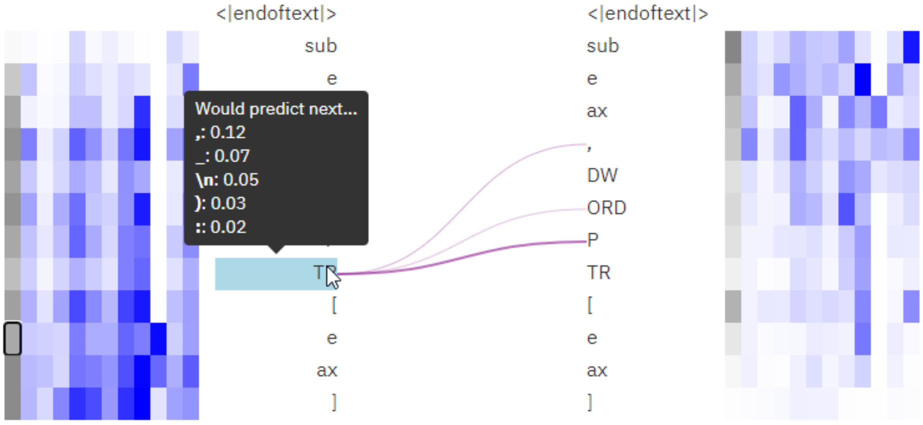


Figure 3.6: In layer-7, head-1, all relation of sentence's tokens

On the left-hand side of Figure 3.5, we see all relationships of tokens on 12-head (multi-head) columns employing a self-attention mechanism. In all layers, tokens on the left multi-heads attend to tokens on the right multi-heads, as in layer-7. Figure 3.6 shows the relationship of tokens on the first left-head-attention of the architecture and

[16] Related link: https://huggingface.co/exbert/?model=gpt2



Would predict next...
,: 0.12
_: 0.07
\n: 0.05
): 0.03
:: 0.02

Figure 3.7: In layer-7, head-1, all probabilities of "TR" calculate with tokens on the right and attend to tokens on the right

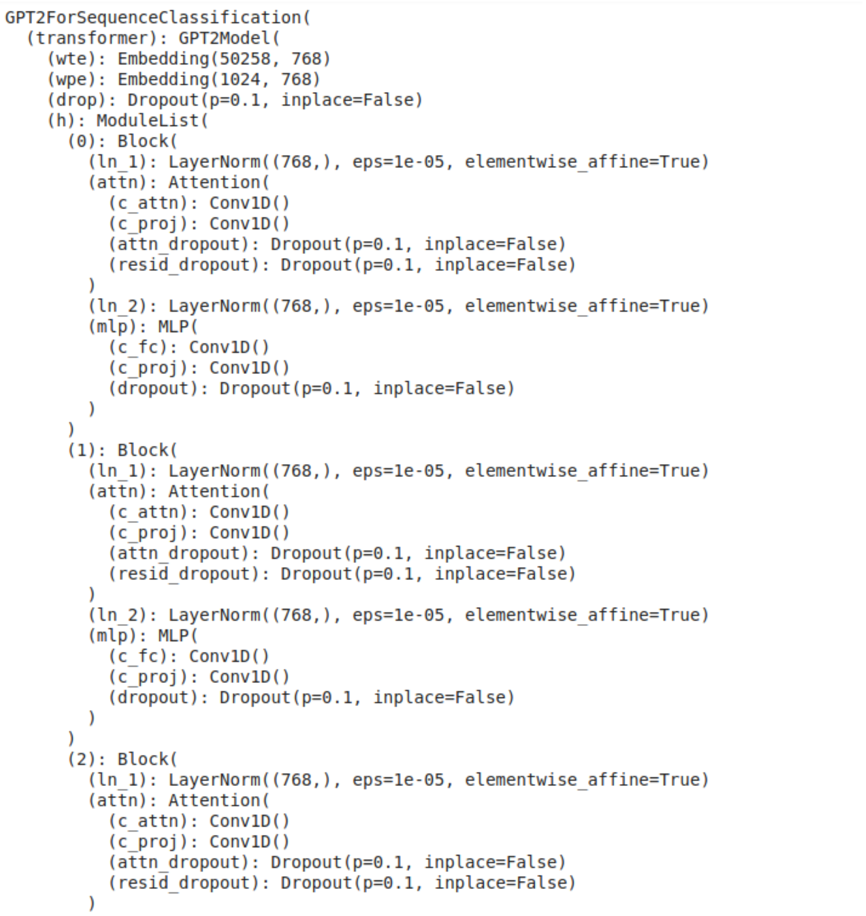
attend to the first right-head-attention of the architecture. In Figure 3.7, the architecture calculates the relation of "TR" with other tokens on the first left-head-attention and accompanies results on the first right-head-attention. Therefore, the architecture also calculates the probabilities of previous tokens of "TR" and probabilities of next tokens of "TR", such as the probability of "," next token of "TR" is 0.12 and probability of "_" next token of "TR" is 0.07.

Furthermore, we constructed binary classification model by adding a classification layer to the *GPT2Model*, known as *GPT2ForSequenceClassification*, as in Figure 3.8. The new model's configuration file was the config.json, and the model initialized with the pytorch_model.bin loading weights associated with the pre-trained model. Next, we trained the model with word embedding and positional embedding obtained from the tokenizer. The classification layer was a densely connected layer with a single output node. To calculate weights during the training, every deep learning model needs a loss function and an optimizer. Since *GPT2ForSequenceClassification* focuses on classifying samples into two categories and model's output is a probability (a single-unit layer with a *sigmoid* activation), *GPT2ForSequenceClassification* is used as the BinaryCrossentropy[17] loss function. Lastly, we configured the model to use an optimizer. We defined optimizer with adamw implementing adam algorithm with weight decay fix in Pytorch. Weight decay is used to prevent overfitting, which keeps the weights as small as possible and prevents the weights from growing out of control. Thus, the network avoids exploding gradients. Before the training phase, we created the learning rate scheduler (Mihaila, 2020).

Splitting data into training, validation and testing sets

In this step, we separated the dataset into three parts named training, validation, and testing to use during the training and testing process. When we split data into three parts, we preferred the accepted common opinion: separating 60% of the dataset for

[17] BinaryCrossEntropy: BinaryCrossEntropy is used to compute the cross-entropy loss between true labels and predicted labels



```
GPT2ForSequenceClassification(
  (transformer): GPT2Model(
    (wte): Embedding(50258, 768)
    (wpe): Embedding(1024, 768)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0): Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): Attention(
          (c_attn): Conv1D()
          (c_proj): Conv1D()
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): MLP(
          (c_fc): Conv1D()
          (c_proj): Conv1D()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (1): Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): Attention(
          (c_attn): Conv1D()
          (c_proj): Conv1D()
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): MLP(
          (c_fc): Conv1D()
          (c_proj): Conv1D()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (2): Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): Attention(
          (c_attn): Conv1D()
          (c_proj): Conv1D()
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
```

Figure 3.8: Sample layers of GPT2ForSequenceClassification model

training, 20% of the dataset for validation, and 20% of the dataset for testing. Up to that point, we performed operations for taking the data from text files into python data structures and preparing the data for the training and testing phase.

Parameters for Training and Testing Processes

In order to find the best values for the parameters' learning rate for adamw and epoch in the language model task, we tried several different values. So, we trained two, three, and four epochs. In three epochs, the model found better accuracy and lower loss value than in two epochs. Also, accuracy and loss results were almost the same for three epochs and four epochs using learning rate value 2e-5, as the results shown in Table 3.4.

Hence, we decided epochs number is three because of reducing the training time. While we examined learning rate values, we decided learning rate value was 2e-5 (e=2.71828). As we selected the learning rate to 3e-5 or 5e-5, network accuracy decreased, and loss results increased as the results shown in Table 3.5.

After the experiments, the binary classification model achieved 85.4% accuracy with a validation loss of 0.37708.

Table 3.4: The effects of epochs on validation losses.

Epochs	Loss Values	Accuracy %
1.	0.40393	0.83902
2.	0.39112	0.84732
3.	0.37708	0.85402
4.	0.37691	0.85407

Table 3.5: The effects of Learning rate on validation losses

Learning rate	Loss Values	Accuracy %
5e-5	0.42397	0.79802
3e-5	0.39819	0.82732
2e-5	0.37708	0.85402

3.4 Summary

In this section, we presented the technical specifications of our methodology, particularly our approach, the dataset collection process, the dataset format, the datasets, the neural network training environment, required libraries and modules, the model pipeline, and the parameters used in the model process. Concisely, we built the pre-trained model on assembly code. Then, we used our pre-trained model for the fine-tuning of the binary classification model to achieve better detection. We used NLP language model techniques to classify assembly codes from malicious and benign files. To create language models on assembly codes, we applied GPT-2 based on transformers. We used modules from Pytorch and Hugging Face libraries on the Python programming language and modules from the torch, Transformers, Sklearn, and Matplotlib libraries to implement our approach. Lastly, we presented the values tried on the parameters required in the training and testing process.

CHAPTER 4

RESULTS

In this chapter, we present the results of the training and detection performances the binary classification model fine-tuning with our pre-trained model and Generative Pre-trained Transformer 2's (GPT-2's) pre-trained model.

4.1 Evaluation Criteria

After creating model, we investigated how successful model was. Therefore, we evaluated the models' performances based on the F1 score and Accuracy, following the common practice in the previous work. F1 score and Accuracy, computed on confusion matrices are among the most popular chosen metrics in binary classification tasks. F1 provides a single score that balances both the concerns of Precision and Recall in one number (Hand et al., 2021). Precision and Recall values are calculated according to Table 4.1. The number of True Positive circumstances and the number of False Negative circumstances are TP, FN respectively. FP is the number of False Positive circumstances, and TN is the number of True Negative circumstances.

Table 4.1: The Confusion matrix for reference

	Actual Label		
Predicted	True Positive(TP)	False Negative(FN)	**Recall**
Label	False Positive(FP)	True Negative(TN)	
	Precision		**F1**

Precision measures the ratio of accurately recognized positive cases against all positive predicted cases. It is formulated as :

$$Precision = \frac{TP}{TP + FP}$$

Recall measures the ratio of accurately recognized positive cases from all actual positive cases. It is formulated as :

$$Recall = \frac{TP}{TP + FN}$$

F1 score is the harmonic mean of Precision and Recall. Hence, this score takes both false positives and false negatives into account. F1 is usually more beneficial if you have an uneven class distribution, whereas Accuracy works properly if false negatives and false positives have similar costs.

$$F1 = \frac{(2 * Precision * Recall)}{Precision + Recall}$$

Accuracy is the ratio of the number of accurately classified samples to the overall number of samples.

$$ACC = \frac{(TP + TN)}{TP + TN + FP + FN}$$

4.1 Evaluation of the Pre-trained Model

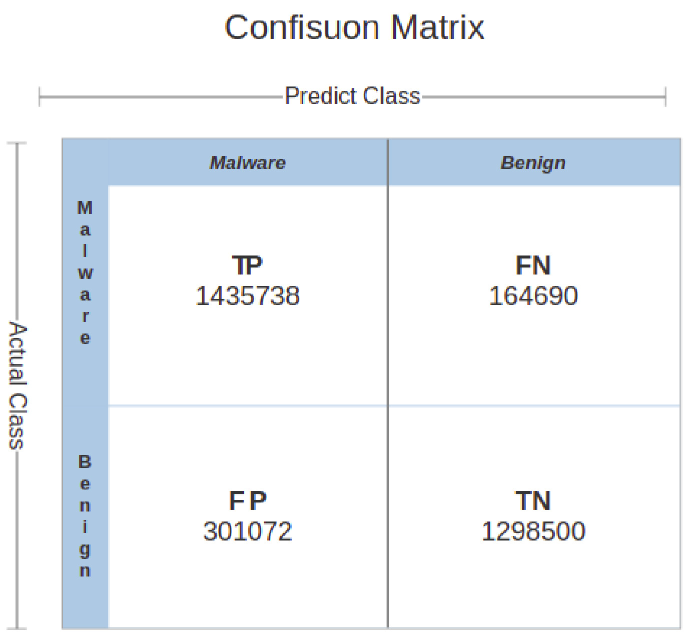
We conducted a total of 11 experiments for the binary classification model. Initially, we experimented with three values for minimum frequency and three values for epochs for creating pre-trained models. Therefore, we obtained six different pre-trained models. Next, we fine-tuned the binary classification models with these pre-trained models. In addition, we experimented with four values for epochs, three values for learning rate in these binary classification models. The number of correctly and incorrectly classified samples obtained is displayed in a confusion matrix (Figure 4.1) for the binary classification model with the best detection performance.

In the confusion matrix, the number of true negatives TN refers to accurately recognized instructions as benign, whereas TP's number of true positives refers to accurately recognized instructions as malicious. The number of false positives FP displays benign instructions recognized as malicious. In contrast, the number of false negatives FN displays malicious instructions recognized as benign.

All scores are in Table 4.2. Precision is as 82.6%, and Recall is as 89.7%. F1 rate is as 86% and Accuracy rate is as 85.4%.

4.2 Evaluation of GPT-2's Pre-trained Model

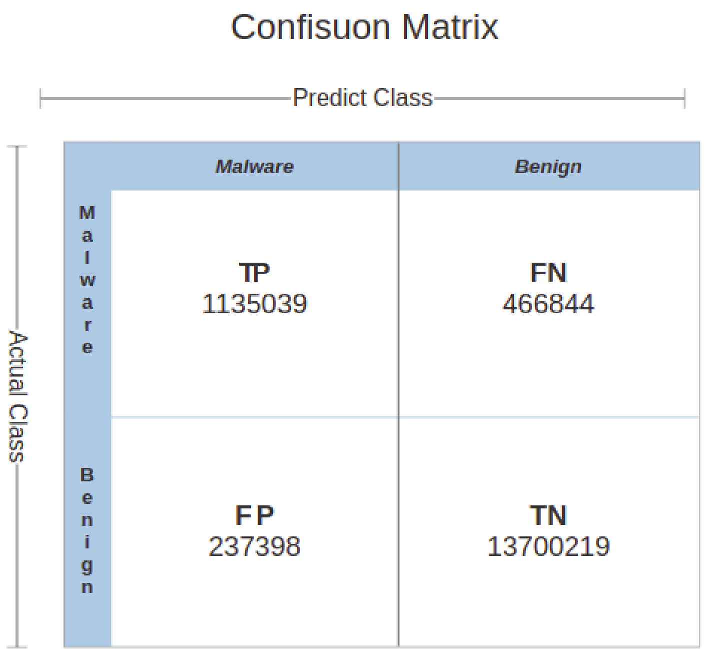
We downloaded the 'gpt2' pre-trained model from the Hugging Face library for fine-tuning the binary classification model. We used the same hyperparameters given the best results on our pre-trained model. Thus, we conducted an experiment (Figure 4.2).



Confisuon Matrix

		Predict Class	
		Malware	Benign
Actual Class	Malware	TP 1435738	FN 164690
	Benign	FP 301072	TN 1298500

Figure 4.1: Confusion matrix of our pre-trained model where TN is the number of true negatives, FN is the number of false negatives, FP is the number of false positives, and TP is the number of true positives



Confisuon Matrix

		Predict Class	
		Malware	Benign
Actual Class	Malware	TP 1135039	FN 466844
	Benign	FP 237398	TN 13700219

Figure 4.2: Training with GPT-2 pre-trained model

Table 4.2: F1 Score Calculation of Binary Classification model fine-tuned with our pretrained model

Performance Metrics	Formules	Results
Precision	$\frac{TP}{TP+FP}$	$\frac{1435738}{1435738+301072} = 0.826$
Recall	$\frac{TP}{TP+FN}$	$\frac{1435738}{1435738+164690} = 0.897$
F1	$\frac{2*Precision*Recall}{Precision+Recall}$	$\frac{1.4818}{1.723} = 0.86$

Precision is as 82.7%, and Recall is as 70.8%. F1 rate is as 76.2% and Accuracy rate is as 78.3%.

4.3 Comparison of the models

The most significant information that we can infer from Table 4.3 is that the specially pre-trained model outperforms GPT-2's pre-trained model. The main difference between two proposed model is the pre-trained models. The first binary classification model is fine-tuned with our pre-trained model. The second one is fine-tuned with

Table 4.3: Comparison of models based on different pretrained models

Model	Precision (%)	Recall (%)	F1 (%)	ACC (%)
Our custom pre-trained	82.6	89.7	86	85.4
gpt2 pre-trained	82.7	70.8	76.2	78.3

gpt2, which is a GPT-2's pre-trained model. Both of them are fed with assembly instructions as sentences and both have the same hyperparameters. The first model achieves an 86% F1 score to include meaningful information and patterns. Nevertheless, the second architecture also achieves an 76.2% F1 score. In summary, the results of the final experiments on the two models suggest that fine-tuning with assembly language parameters(our pre-trained model) seems more efficient than GPT-2's pre-trained model on detecting malware. Despite the insufficient GPU memory, binary classification looks to be successful with 86% F1 score, as well.

4.4 Discussion

However, these techniques face two challenges. The first challenge is to model the full semantics behind malware or benign software assembly code. The second is to provide explainable results while keeping efficient detection performance. Models have various approaches identifying different feature representations and different kinds of classification algorithms for these challenges. The most studied and used classification algorithms are listed as random forest (RF), support vector machine (SVM), and decision tree (DT) algorithms. Those algorithms are designed as shallow learning architectures that learn from pre-defined features. Though they are successful in malware detection, shallow learning architectures are still insufficient in malware detection since feature engineering, feature learning, and feature representation require data by humans. Therefore, we need high-level machine learning (ML) architectures such as deep learning (DL) to utterly avoid the feature engineering phase. The main strength of deep learning architectures is understanding the meaning of data regardless of its large amounts. Furthermore, it can automatically fine-tune the derived meaning with new data without expert knowledge, automatically providing detection tasks.

In academic studies, DL architectures are currently researched to identify malicious and benign software, and classified malicious files according to their corresponding families. Researchers propose various architectures and methods, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and attention mechanisms. CNNs, the first architecture of DL, are popularly used in advance malware detection methods. In most of the CNN methods proposed in the literature, such as (Krčál et al., 2018), and (Kumar et al., 2018) opcode sequences or assembly instructions of malicious and benign software are converted into images. Next, the neural network is trained on those images. Recurrent neural networks (RNNs) work on sequential data to extract patterns that are serving the data. It delivers better on tasks including long sequential data such as speech recognition and natural language classification. For instance, (Jha et al., 2020) shows that the RNN model is effectively detecting malware with opcodes sequence. Another long sequential data processor is long short-term memory (LSTM), a specialized RNN architecture. The studies in the literature that employs LSTM such as (Lu, 2019) for malware detection purposes focus on opcode sequences instead of the whole assembly code. The transformer processes data significantly on short-text and defines a new state-of-the-art with an attention mechanism that provides global dependencies between input and output. The work in (M. Li et al., 2021) used benign and malicious assembly code obtained from the static content of an executable. Their model, Interpretable MAlware Detector (I-MAD) based on transformers, combined a network component called the Galaxy Transformer network that recognized assembly code at the basic block(a sequence of assembly instructions), assembly function (a set of basic blocks), and executable levels (a sequence of bytes). Its feed-forward neural network provided interpretations for its detection results by quantifying the impact of each feature for the prediction. In another work (Rahali & Akhloufi, 2021) employed a transformer-based model "Bidirectional Encoder Representations from Transformers (BERT)" for malware detection. Researchers focused on the source code of Android applications with static analysis to classify different malware categories. BERT's pre-trained models use a compound of masked language model objective and next sentence prediction on a large corpus, including the Toronto Book Corpus and Wikipedia. However, we used decoder transformers "Generative Pre-trained Transformer 2 (GPT-2)" instead of encoder transformers BERT because of its success in predicting the next word given the previous words. Our decoder-based, context-aware networks use the attention mechanism layers to retain contextual characteristics from a natural language sentences perspective. We quantified assembly instructions as sentences. We constructed our pre-trained model based on GPT-2 architecture to learn syntax and semantics representations of opcodes and operands in assembly instructions. We also built a binary classification model based on GPT-2. This model was fine-tuned with our pre-trained model to detect malware,namely our model. We then built another binary classification model based on GPT-2 and fine-tuning it with GPT-2's pre-trained model *gpt2*, namely gpt2. Since BERT is commonly used in literature, we next created a binary classification model based on BERT and fine-tuned it with the pre-trained model *bert-base-uncase*, namely bert-base-uncase. Finally, we compared these three models. As shown in Table 4.4, the binary classification model based on GPT-2 that was fine-tuned with our pre-trained model gave better results than others.

Table 4.4: Evaluation of our proposed methods with Transformers-based model

Pre-trained Models	Data Format	ACC (%)
bert-base-uncase	Assembly Instructions	77.6
gpt2	Assembly Instructions	78.3
our model	Assembly Instructions	85.4

4.5 Open Problems

While studying the recent articles and working on our study, we have noticed that some points can be considered as challenges in the malware detection domain. The first challenge is the datasets used in the research. Since the studies related to the malware domain do not have common or benchmark datasets, each research tried to create their dataset as described in Section 3.2. We got robust insights using our dataset, which included multiple data sets, but there will always be a limit to what we can infer from an experimental study. For instance, multiple data sets may not be independent and may have similar biases. There is also the quality issue, which is a particular issue in deep learning datasets. The need for the quantity of data limits the amount of quality checking that can be done. The issues that need special attention are briefly summarized in the recently published article (Lones, 2021) on this aspect. Our experiments tried to ensure that our models produced healthy and comparable results by changing the ratios and places of the training set, test set, and validation sets. Firstly, while developing the model, we used the same seed value as the training set, test set, and validation set to separate the dataset. Thus, we ensured that we got the same environment for reproducibility[1] with set_seed(123)[2]. After our model matured, we used different seed values to obtain different dataset samples as training and validation sets.

We experimented with four different values (42, 82, 123, 176) for seed value and inspected the change in the accuracy of the binary classification model as in table 4.5. We can see that results are around 85.4% accuracy. Moreover, 0.35% difference in accuracy could be entirely explained by just a seed difference.

Table 4.5: Seed value effect on performance

Experiments	seed values	F1(%)	Acc(%)
1.	42	0,860028109	85.5
2.	82	0,860001704	85.1
3.	123	0,860011608	85.4
4.	176	0,859001307	84.7
		Mean F1:0.859762432	Mean:Acc:85,175

Another challenges, the authors describe multiple ways to extract features and apply multiple machine learning models in the literature. For this reason, we described and used transformers-based models. We adopted the Deep Neural network model, i.e., GPT-2 models fine-tunning with our custom pretrained model and gpt-2 GPT-2's pre-trained model, and BERT model fine-tuning with bert-base-uncase. In conclusion, we obtained remarkable results with an 86% F1 score and 85.4% accuracy in literature.

[1] https://en.wikipedia.org/wiki/Reproducibility

[2] https://huggingface.co/transformers/main_classes/trainer.html

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Threats from malicious software have grown day by day in terms of complexity and number. Hence, researchers have developed automated methods to detect and classify malware in order to defend information systems instead of manually analyzing them in a time-consuming effort. In early times, machine learning-based malware analysis methods solved this issue using statistical methods and machine learning (ML) methods. However, those algorithms do not afford fully automated methods. In contrast, deep learning (DL) neural networks mimic the learning process better and provide smarter and faster tools. Although deep neural networks have mainly resulted in notable performance gains for learning, a careful comparison between different approaches may be challenging. In general, since all approaches focus on long-term dependency,time and memory size become critical for long-term dependency. Transformers-based models have reasonable solutions for these challenges. They especially achieve state-of-the-art results in natural language understanding and produce efficient solutions. Their successful applications are mainly processing short-texts on multiple head architectures, each with a single self-attention mechanism. The multi-head attentions provide global dependencies between input and output with the position-wise feed-forward networks layers. Moreover, short-text also finds a much better solution for vanishing/exploding gradient problems. In addition, transformer-based architectures' (e.g. XLnet, GPT-2, GPT-3) success results from mainly being pre-trained models.

In this study, we proposed a generative approach to classify malicious and benign software. We worked on assembly language since the assembly code provides accurate information for obtaining the critical coding patterns. We implemented our strategy on static analysis data and focused on opcodes and operands instead of just opcodes due to the attention mechanism. First, we modeled to grab the full semantics behind the assembly code with a pre-trained model based on GPT-2. Next, we provided explainable results while keeping effective detection performance with labeled code. We modeled binary classification with the transformers-based model GPT-2 with malicious and benign assembly codes. Furthermore, we fine-tuned the binary classification model with the pre-trained model to improve the detection performance. We selected optimum parameter values for our neural network architectures based on our experimental results. The resulting accuracy rate (85.4%) shows that it is possible to classify malicious and benign assembly codes by GPT-2 with the pre-trained model.

We experimented with that binary classification model fine-tuned with our pre-trained model based on GPT-2 and GPT-2 architecture's pre-trained model gpt2. As a result of this approach, we achieved more successful results with our pre-trained model. In addition, in 3 epochs, we trained our binary classification model with 8 million lines and model accuracy was 82.1%. While we trained it with 12 million lines, it was 83.9%. When we fed it with 16 million lines, the success of the prediction made by our binary classification model on data has reached 85.4%. Therefore, we believe that our models may give better results with more powerful memory and GPU.

5.2 Limitations and Future Work

Transformer-based architectures exhaust the computational and memory resources too much since transformer-based architectures yield the best result when they can replicate the data across GPUs. The significant limitation of the study is not having adequate memory resources.

Some aspects of this study can be further improved and optimized in the future. Future research should train in transformers-based architecture we choose and different transformer-based architectures, such as GPT-3 and transformer XL, with more efficient memory resources for advancing detection. Future research should address improvements in the data processing pipeline, developing an API with the disassembler of x86 Windows executable files to automatically disassemble when encountering a new malware. The models update their parameters on multiple GPUs in parallel with the API. Thus, there is no need for human intervention. Future research should also address moving our current detection process from the code level to the file level, and should also apply our proposed method for classifying different types of malware, such as worms, trojan horses, at the OS level both for desktop and mobile operating systems. We also may focus on specific malware like spyware and crypto miners to obtain better detection results. The neural network architecture, GPT-2, allows us to create black-box models because of the incomprehensible internal logic of the hidden layers. So, as in every other study using deep neural networks, having a black box at the model level limits this study. While we can always search for better hyperparameters, there is no common method to do this. In this study, while we were designating parameters required for the language model, we tried several values and picked the best ones that showed the best performance on our data. However, in deep learning research, different datasets can result in different outcomes between similar studies through these assumptions. Hence, opaque hyperparameters are limited to this study. As a result, the nature of deep neural network architectures poses a limitation. Moreover, in deep learning, the specified parameters' studies might be limited to the dataset used in the corresponding research. In this respect, future research should focus on explainable deep learning. A future success delivered in this subject may also allow us to eliminate such architectural and dataset-related limitations.

REFERENCES

Acarturk, C., Sirlanci, M., Balikcioglu, P. G., Demirci, D., Sahin, N., & Kucuk, O. A. (2021). Malicious code detection: Run trace output analysis by lstm. *IEEE Access*, 1–1. https://doi.org/10.1109/access.2021.3049200

Akhtar, Z., Micheloni, C., & Foresti, G. L. (2015). Biometric liveness detection: Challenges and research opportunities. *IEEE Security & Privacy*, *13*(5), 63–72. https://doi.org/10.1109/msp.2015.116

Al-Rfou, R., Choe, D., Constant, N., Guo, M., & Jones, L. (2018). Character-level language modeling with deeper self-attention. *arXiv:1808.04444 [cs, stat]*. https://arxiv.org/abs/1808.04444

Avgerinos, T., Cha, S. K., Hao, B. L. T., & Brumley, D. (2011). Aeg: Automatic exploit generation. *kilthub.cmu.edu*. https://doi.org/10.1184/R1/6468296.v1

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. https://arxiv.org/abs/1409.0473

Bayer, U., Comparetti, P., Hlauschek, C., Kruegel, C., & Kirda, E. (2009). *Scalable, behavior-based malware clustering*. https://sites.cs.ucsb.edu/~chris/research/doc/ndss09_cluster.pdf

Becker, C., Hahn, N., He, B., Jabbar, H., Plesiak, M., Szabo, V., To, X.-Y., Yang, R., & Wagner, J. (2020). *Modern approaches in natural language processing*. https://compstat-lmu.github.io/seminar_nlp_ss20/transfer-learning-for-nlp-i.html

Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C., Ca, J., Kandola, J., Hofmann, T., Poggio, T., & Shawe-Taylor, J. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*, 1137–1155. https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

Bilar, D. (2007). Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, *1*(2), 156. https://doi.org/10.1504/ijesdf.2007.016865

Brownlee, J. (2020). What is argmax in machine learning? https://machinelearningmastery.com/

Brumley, D., Poosankam, P., Song, D., & Zheng, J. (2008). Automatic patch-based exploit generation is possible: Techniques and implications. https://doi.org/10.1109/SP.2008.17

Burmester, S. (2020). The rising cost of a data breach in 2020. https://www.ibm.com/blogs/ibm-anz/the-rising-cost-of-a-data-breach-in-2020/

Contributors, W. (2018). Artificial neural network. https://en.wikipedia.org/wiki/Artificial_neural_network

Contributors, W. (2019). Linguistics. https://en.wikipedia.org/wiki/Linguistics

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, *44*(2), 1–42. https://doi.org/10.1145/2089125.2089126

Eisenstein, J. (2019). *Introduction to natural language processing*. The Mit Press.

Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, *05*(02), 56–64. https://doi.org/10.4236/jis.2014.52006

Gibert, D., Mateu, C., & Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, *153*, 102526. https://doi.org/10.1016/j.jnca.2019.102526

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *"sequence modeling: Recurrent and recursive nets" in deep learning*. The Mit Press.

Hand, D. J., Christen, P., & Kirielle, N. (2021). F*: An interpretable transformation of the f-measure. *Machine Learning*, *110*, 451–456. https://doi.org/10.1007/s10994-021-05964-1

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *arXiv:1603.05027 [cs]*. https://arxiv.org/abs/1603.05027

Hendrycks, D., & Gimpel, K. (2020). Gaussian error linear units (gelus). *arXiv:1606.08415 [cs]*, *4*. https://arxiv.org/abs/1606.08415

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Horan, C. (2021). Unmasking bert: The key to transformer model performance. https://neptune.ai/blog/

Jha, S., Prashar, D., Long, H. V., & Taniar, D. (2020). Recurrent neural network for detecting malware. *Computers & Security*, 102037. https://doi.org/10.1016/j.cose.2020.102037

Khan, R. U., Zhang, X., & Kumar, R. (2018). Analysis of resnet and googlenet models for malware detection. *Journal of Computer Virology and Hacking Techniques*, *15*(1), 29–37. https://doi.org/10.1007/s11416-018-0324-z

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv.org*. https://arxiv.org/abs/1412.6980

Krčál, M., Ondřejšvec, Jašek, O., & Avast, M. (2018). *Deep convolutional malware classifiers can learn from raw executables and labels only*. https://openreview.net/pdf?id=HkHrmM1PM

Krishna, A., Santra, B., Bandaru, S. P., Sahu, G., Sharma, V. D., Satuluri, P., & Goyal, P. (2018). Free as in free word order: An energy based model for word segmentation and morphological tagging in sanskrit. *arXiv:1809.01446 [cs]*, *2*. https://arxiv.org/abs/1809.01446

Kumar, R., Xiaosong, Z., Khan, R. U., Ahad, I., & Kumar, J. (2018). Malicious code detection based on image processing using deep learning. *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence - ICCAI 2018*. https://doi.org/10.1145/3194452.3194459

Lam, W. (2021). Seem 5680. https://www1.se.cuhk.edu.hk/~seem5680/

Lee, Y., Kwon, H., Choi, S.-H., Lim, S.-H., Baek, S. H., & Park, K.-W. (2019). Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, *9*(19), 4086. https://doi.org/10.3390/app9194086

Lemos, R. (2021). Ransomware, phishing will remain primary risks in 2021. https://www.darkreading.com/threat-intelligence/ransomware-phishing-will-remain-primary-risks-in-2021/d/d-id/1340256

Li, M., Fung, B. C., Charland, P., & Ding, S. H. (2021). I-mad: Interpretable malware detector using galaxy transformer. *Computers & Security*, *108*, 102371. https://doi.org/10.1016/j.cose.2021.102371

Li, X., Yu, Q., & Yin, H. (2021). Palmtree: Learning an assembly language model for instruction embedding. *arXiv:2103.03809 [cs]*. https://doi.org/10.1145/3460120.3484587

Lin, Y.-D., Lai, Y.-C., Lu, C.-N., Hsu, P.-K., & Lee, C.-Y. (2015). Three phase behavior based detection and classification of known and unknown malware. *Security and Communication Networks*, *8*(11), 2004–2015. https://doi.org/10.1002/sec.1148

Liu, F., Ren, X., Zhang, Z., Sun, X., & Zou, Y. (2021). Rethinking skip connection with layer normalization in transformers and resnets. https://arxiv.org/pdf/2105.07205.pdf

Lones, M. (2021). How to avoid machine learning pitfalls: A guide for academic researchers. *arXiv:2108.02497 [cs]*. https://arxiv.org/abs/2108.02497

Lu, R. (2019). Malware detection with lstm using opcode language. *arXiv:1906.04593 [cs]*. https://arxiv.org/abs/1906.04593

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. https://arxiv.org/abs/1508.04025

Mihaila, G. (2020). Gpt2 finetune classification george mihaila. https://gmihaila.github.io/tutorial_notebooks/gpt2_finetune_classification/

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. https://arxiv.org/pdf/1310.4546.pdf

M.K., H. (2019). Backpropagation step by step. https://hmkcode.com/

Moskovitch, R., Stopel, D., Feher, C., Nissim, N., & Elovici, Y. (2008). Unknown malcode detection via text categorization and the imbalance problem. *IEEE Xplore*. https://doi.org/10.1109/ISI.2008.4565046

Naseer, M., Khan, S., Hayat, M., Zamir, S. W., Khan, F. S., & Shah, M. (2021). Transformers in vision: A survey. *arXiv:2101.01169 [cs]*. https://arxiv.org/abs/2101.01169

Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware images. *Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec '11*. https://doi.org/10.1145/2016904.2016908

Openai, A., Openai, K., Openai, T., & Openai, I. (2018). *Improving language understanding by generative pre-training*. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

Pai, S., Troia, F. D., Visaggio, C. A., Austin, T. H., & Stamp, M. (2016). Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques*, *13*, 95–107. https://doi.org/10.1007/s11416-016-0265-3

Pei, K., Xuan, Z., Yang, J., Jana, S., & Ray, B. (2021). Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv:2012.08680 [cs]*. https://arxiv.org/abs/2012.08680

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. https://arxiv.org/abs/1802.05365

Platen, P. V. (2020). Transformer based encoder and decoder models. https://huggingface.co/blog/

Preda, M. D., Christodorescu, M., Jha, S., & Debray, S. (2008). A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, *30*(5), 1–54. https://doi.org/10.1145/1387673.1387674

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Rahali, A., & Akhloufi, M. A. (2021). Malbert: Using transformers for cybersecurity and malicious software detection. *arXiv:2103.03806 [cs]*. https://arxiv.org/abs/2103.03806

Rosenthal, M. (2020). Must-know phishing statistics: Updated 2020. https://www.tessian.com/blog/phishing-statistics-2020/

Samani, R. (2021). https://www.mcafee.com/enterprise/en-us/lp/threats-reports/apr-2021.html

Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., & Bringas, P. G. (2010). Idea: Opcode-sequence-based malware detection. *Lecture Notes in Computer Science*, *5965*, 35–43. https://doi.org/10.1007/978-3-642-11747-3_3

Santos, I., Sanz, B., Laorden, C., Brezo, F., & Bringas, P. G. (2011). Opcode sequence based semi supervised unknown malware detection. *Computational Intelligence in Security for Information Systems*, *6694*, 50–57. https://doi.org/10.1007/978-3-642-21323-6_7

Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv.org*. https://arxiv.org/abs/1508.07909

Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., & Elovici, Y. (2012). Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, *1*(1). https://doi.org/10.1186/2190-8532-1-1

Shieber, S., & Rush, A. (2018). The annotated transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Shimon, O. (2021). https://www.deepinstinct.com/2021/02/11/cyber-threat-report-on-2020-shows-triple-digit-increases-across-all-malware-types/

Souri, A., & Hosseini, R. (2018). A state of the art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, *8*(1). https://doi.org/10.1186/s13673-018-0125-x

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(56), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

Tay, Y., Research, G., Dehghani, M., Bahri, D., & Metzler, D. (2020). *Efficient transformers: A survey efficient transformers: A survey*. https://arxiv.org/pdf/2009.06732.pdf

Ucci, D., Aniello, L., & Baldoni, R. (2019). Survey of machine learning techniques for malware analysis. *Computers and Security*, *81*, 123–147. https://doi.org/10.1016/j.cose.2018.11.001

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. https://arxiv.org/abs/1706.03762

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., & et al. (2020). Huggingface's transformers: State-of-the-art natural language processing. *arXiv:1910.03771 [cs]*. https://arxiv.org/abs/1910.03771

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T.-Y. (2020). On layer normalization in the transformer architecture. *arXiv:2002.04745 [cs, stat]*. https://arxiv.org/abs/2002.04745

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., & Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv.org*. https://arxiv.org/abs/1906.08237

Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys*, *50*(3), 1–40. https://doi.org/10.1145/3073559

Yewale, A., & Singh, M. (2016). Malware detection based on opcode frequency. https://doi.org/10.1109/ICACCCT.2016.7831719

Zhang, H., Xiao, X., Mercaldo, F., Ni, S., Martinelli, F., & Sangaiah, A. (2019). Classification of ransomware families with machine learning based on n gram of opcodes. *Future Generation Computer Systems*, *90*, 211–221. https://doi.org/10.1016/j.future.2018.07.052